ASCII SOFTWARE SCIENCE *Language 5*

改訂新版

C++言語入門

STEPHEN C. DEWHURST, KATHY T. STARK 共著

小山裕司 訳



アスキー出版局

SOFTWARE SCIENCE

SOFTWARE SCIENCE

改訂新版

C++言語入門

STEPHEN C. DEWHURST, KATHY T. STARK 共著

小山裕司 訳

アスキー出版局

■商　標
本書で登場するシステム名、製品名は一般に各開発メーカーの登録商標です。なお本書では、TM、®マークは明記しておりません。

表紙デザイン　花本浩一
本文イラスト　大竹伸一

PROGRAMMING IN C++

Second Edition

Stephen C. Dewhurst • Kathy T. Stark

序

本書の初版が1989年に出版されて以来，C++の利用者コミュニティは急激に成長を続けている。ほとんど毎日といってよいほど，新しいプログラミングテクニックが考案され，またC++自体も成長を続けている。1989年当時のC++コミュニティの様子を特徴付けていた「想像力に富んだ意見交換」は今日にいたっても衰えを知らず，さらに米国規格協会(ANSI)と国際標準化機構(ISO)の援助のもと，標準規格の制定も進んでいる。

本の著者またコンパイラ開発者としての，現在の私達の仕事は，当時ほど楽ではない。成長中の言語を解説するにあたって，新しい標準規格を細部まで解説し，目立ったプログラミングテクニックのすべてに言及しようという誘惑にかられる。しかし，本書の初版への読者からの反応は，言語機能よりもプログラミングの解説を重視した私達の選択は正しいものであったことを示している。ゆえに，この第2版でも私達は，プログラミングパラダイムと，それらをサポートする言語機能の議論を同時に展開し，読者が単にC++言語を構文的に理解することを越え，C++を柔軟で効果のあるプログラミングツールとして使うことができるようにさせたいと考えている。

この第2版の執筆にあたり，多数の方々の協力を得た。彼らすべてに感謝したい。特に，Christopher Aoki氏が与えてくれた援助と忍耐力と励ましに感謝したい。Sarah Hewins女史は，草稿のいくつかの版を読み，鋭い指摘を与えてくれた。彼女が与えてくれた援助と催促に特に感謝したい。

Clovis Tondo氏は，この第2版の製作の隠れた推進力であった。本書を執筆していた数年の間，彼は私達に動機付け，励ましを与えてくれた。また技術的に査読を行い，

さらに私達がちゃんと執筆を続けるようにときどき催促してくれもした。

初版の日本語への翻訳者である小山裕司氏は，第１章のワイド文字に関する資料と例を提供してくれ，また，査読者としても援助してくれた。私達は彼が与えてくれた貢献と執筆期間を通じての励ましに感謝したい。

S.C.D.
K.T.S.

訳者の序

C++は，現在では，事実上の標準プログラミング言語の地位を付き，ANSI/ISOによる標準規格の制定作業も現在規格草案の公式審議段階にある。この言語は，いくつもの概念と特徴が相互に協力し，強めあうようにデザインされている。このため，C++による恩恵を本当に得るには，単に言語仕様を学んだだけでは不充分で,「プログラミング」の手法を学ぶ必要がある。

本書は，Stephen C. Dewhurst 氏と Kathy T. Stark 女史による『Programming in C++, Second Edition』の翻訳である。原書の初版は，C++の「プログラミング」の解説書として非常に高い評価を得ている。

この第2版の特徴は，最新の標準規格案への対応である。本書の初版は，C++がまさに発展している最中に書かれたものであるが，現在でもC++のプログラミングを解説した良書として依然として有効である。しかし，6年の歳月を経て，C++には新しい機能(テンプレートと例外処理)が追加され，また昔からあった機能も細部が変更されているものがいくつかある。さらに，初版で解説してあったプログラミングテクニックの中には，現在でも通用するが，新しい機能を使うことで，よりエレガントに表現できるものもある。今回の第2版では，古びた箇所は修正され，さらに最新のC++での新しい「プログラミング」の解説が追加されている。

訳出にあたっては，原著者の両氏に特に感謝したい。彼らには，執筆と訳出の期間にいくつもの要望と質問に答えていただき，さらに翻訳の原料をいちはやく提供していただいた。また，C++の創始者である Bjarne Stroustrup 博士にも感謝したい。彼に

は本書第2版の噂を教えていただき，さらに翻訳期間にいくつもの質問に答えていただいた。最後に，初版から続いて編集作業をしていただいた株式会社アスキーの金光雅夫氏にも感謝したい。彼には，草稿をていねいに読んでいただき，多くの指摘をいただいた。

1995年9月　小山 裕司

序(初版)

プログラミング言語C++の人気は，高まりつつある。親言語のCは，もともと評価が高い言語であったが，C++にはデータの抽象表現とオブジェクト指向という特徴が追加され，さらに魅力あるものになっている。開発過程では利用者による活発な討論が行われ，多数の想像力に富んだ意見交換の末にC++が形成された。その結果，C++は，さまざまなプログラミング手法やテクニックをサポートする広範囲の機能を備えた言語となった。

言語C++の機能がプログラムの記述にどのような影響を及ぼすかという知識は，C++を理解するための最高の手助けとなるだろう。私達は，C++の形成過程において利用者と実装者という両方の立場からC++の各種の機能の存在意義を知り，実装と利用を通じてC++の進歩に関与するという滅多にない機会を持った。本書が，プログラミングツールとしてのC++を理解し応用するための一助となれば幸いである。

私達は，本書をまだC++が進化している段階で書き下ろした。その間に多重継承のような新しい機能が生まれたり，言語定義の変更や処理系の改良も継続して行なわれており，私達がここで紹介のために使用した処理系は，読者が利用できるバージョンの処理系とは多少だが異なっている可能性もある。それゆえ本書の解説では，異なるバージョンを使っている利用者が混乱しないように，細部をある程度省くことにした。私達は言語のリファレンスマニュアルとか特定の処理系の解説を書くつもりはない。本書では，C++の外観の背後に見え隠れするプログラムデザインの一般的な概念を，プログラマがC++を有効に活用する手助けとして紹介するように心がけた。

本書はC++の開発に携わった多数の方々の助言を受けて完成された。C++のプログ

ラミングパラダイムやプログラミングテクニックの開発に貢献した多数の方々に，ここで感謝しておきたい。彼らの考え方によって大きな影響を受け，その発想の多くが本書の中に実ることになった。Tom Cargill，Jim Coplien，Keith Gorlen，Bill Hopkins，Andy Koening，Jerry Schwarz，Jon Shopiroの諸氏は，こうしたC++の先駆者達の中でも中心的な人物である。Laura Eaves，Phil Brown，Stan Lippmanの諸氏は私達と一緒にC++コンパイラの開発を行なった仲間である。Sarah Hewins，Bob Wilson，Wayne Wolfの諸氏は草稿の段階での検討に加わってくれた。Jeff Monin氏は，いくつかのキーポイントにおいて，時間と専門知識を貸してくれた。こうした同僚達に感謝したい。そして，Prentice-Hall社のJohn Wait氏が，この企画に対して終始変らぬ励ましと指導を与えてくれた。

とりわけ，Brian Kernighan氏に対しては感謝の念を惜しまない。彼は，各稿で不明瞭な部分を明瞭な状態へと導いてくれた。本書を完成することができたのは，ひとえに彼の忍耐と説得のお蔭だといえるだろう。

最後になったが，Bjarne Stroustrup氏はC++を創造しただけでなく，積極的に討論を行いながら，それを豊かなプログラミング言語へと進化させた。彼が，この企画をサポートし，かつ私達と意味のある議論を多数持ってくれたことに感謝したい。

S.C.D.
K.T.S.

監訳者の序(初版)

コンピュータの進化は，急激な速度で進み，とくに，ハードウェアの性能は十数年前とは比較にならないほど優れたものになった。当然のことだが，言語などのソフトウェアも進化が要求される。AT&T ベル研究所の Bjarne Stroustrup 氏がデザインした C++ も，こうした進化の１つである。この C++ は，システム記述言語の傑作である C に,「データの抽象表現」という概念を絶妙のバランスで融合させた言語である。

本書は，AT&T ベル研究所で C++ コンパイラの開発に従事していた Stephen C. Dewhurst 氏と Kathy T. Stark 女史の共著書『Programming in C++』を翻訳したものである。本書が C++ の他の書籍と異なるのは,「プログラミング」に主眼が置かれていることである。C++ は，C に比べて多機能であり，それゆえ，機能(言語仕様)の解説に紙面のほとんどが割かれることが多い。本書は,「言語仕様」の詳細な解説は他の書籍にまかせ，原著者の豊富な知識をもとに，データ抽象のプログラミング，さらにはオブジェクト指向のプログラミングについても，しっかり解説してある。そのため，経験を積んだ利用者にも十分参考になる内容である。また，多数の読者には既知の話題(たとえば，リスト構造やハッシュ表などの解説)と思われることはほとんど省かれている。このことは，本書のもう１つの特徴でもある。あまりに詳細に書かれた書籍は，膨大な量ゆえに，読者の意欲を序々に低下させるが，無駄のない解説を行い，比較的短い量に収まっている本書は，最後まで常に読者を引き付け続けるだろう。

翻訳にあたっては，原著書を注意深く読み，できるだけ分りやすい訳出を心がけたもりであるが，至らない点も多々あることと思われる。読者諸兄のご叱正，ご教示をお願いしたい。

翻訳に際し，多数の方々の協力を得た。東京大学の鈴木慎司氏，斎藤靖氏には，多大な助言を頂いた。日本タイムシェア株式会社の石塚圭樹氏，株式会社アスキーの高橋正和氏，柳楽直樹氏には，本書の草稿を読んで頂き，多数の誤りを指摘して頂いた。東京都立科学技術大学の石島辰太郎教授には，C++の基礎知識を与えて頂いた。最後に株式会社アスキーの金光雅夫氏は，多忙の中を本書の企画を進め，編集の労をとられた。心から感謝の意を表す次第である。

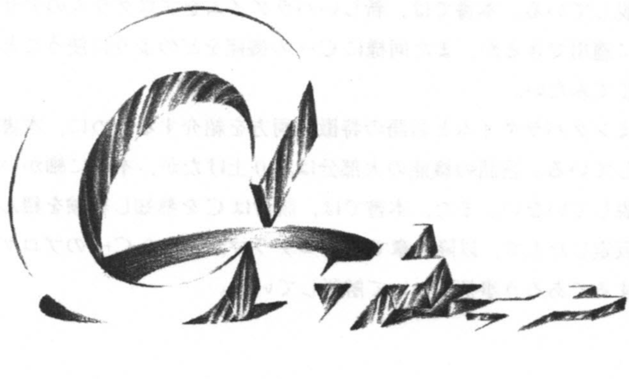
本書が，我が国におけるC++の普及に役立てば，訳者として，これ以上の喜びはない。

1990年6月　小山 裕司

目　次

0

はじめに



C++は，C から派生した汎用プログラミング言語である。C++では，親言語の C にいくつかの特徴が追加された。最も重要な特徴は，データの抽象表現とオブジェクト指向のプログラミングであろう。C++は C の言語仕様のほとんどを受け継いでおり，C の基本データ型，演算，構文規則，プログラム構造に，新しい特徴を見事に融合させている。新たに追加された機能は，新しいプログラミングテクニックをサポートするだけでなく，C から受け継いだ機能も高めている。

プログラミング言語は自然発生するものでなく，実際は人々のプログラミングに対する考え方の変化に従って生まれるものである。こうした思考過程は多数の人々に普及し，文化となることでいくつかのパラダイムとなり，その結果それらをサポートする新しい言語が生み出される。プログラミングパラダイムとは，プログラムのデザインや実装に適用されるさまざまなテクニックを提供するモデルである。これらのテクニックは，プログラミング上の問題に対するデザイン法，抽象表現の技法，プログラムの構成法などに関する議論を生み出す。ある言語の機能によって，そういったテクニックが容易に適用できるならば，その言語は特定のパラダイムをサポートしていることになる。C++の機能は，利用者にいずれか 1 つのパラダイムを強いることなく，ある範囲のテクニックを許している。

プログラミング言語を解説した書籍の多くは，その言語の特徴を教示しはするが，読者が 1 人で新たにプログラミングを試みることまでは面倒をみない。本書が扱う C++は，多くの場合，読者が熟知している言語よりも豊富な内容を持つと思われるため，こうしたアプローチは満足できるものではない。本書は，「C++を使ったプログラミング」について解説している。本書では，新しいパラダイムをプログラムのデザインや実装にどのように適用できるか，また同様に C++の機能をどのように使うことができるかを詳細に論じてみたい。

プログラミングパラダイムと言語の特徴の両方を紹介するために，本書では斬新な方法を採用している。言語の機能の大部分は取り上げたが，本当に細かい部分まではいちいち解説していない。また，本書では，読者は C を熟知し経験を積んだプログラマであると仮定した上で，以降の章で C プログラマが新たな C++のプログラミングに際して直面するであろう事柄について解説している。

0.1 C++言語

C++がCに対して行った主要な追加にクラス型の導入がある。利用者は，クラスを使って，データメンバだけでなく型に作用する関数もまとめた集合データ型を定義することができる。クラスによるデータ隠蔽は，データを抽象表現するための機構を提供し，クラス継承の機構がデータ抽象をオブジェクト指向のプログラミングに拡張する。テンプレート機能は，クラスをパラメタによる総称型にする。利用者定義の演算子関数と型変換は，式にクラス型のオペランドを許し，クラスを現行の型システムに組み込むことを可能にしている。

クラスには，データ構造の動的な記憶管理を可能にする機能もある。また，記録領域の動的な割り当てや解放のための汎用の演算子も提供されている。利用者は，これらを組み合わせて，特定のアプリケーションあるいは問題領域に適した独自の記憶管理機構を構築することができる。

C++では，データ構造を構築するテクニックをサポートする機能のほかに，関数まわりの機能も強化されている。C++の関数宣言には引数の型情報がかならず含まれている。これは，引数の型を検査することと，1つの関数名を複数の実装に多重定義することを可能にする。関数の引数宣言には，関数の呼び出し時に実引数を与えなかった時のためのデフォルト値を与えることができる。また，関数の値呼び出しのほかに，リファレンス型引数によるリファレンス呼び出しも可能である。例外処理は，特別の処理を要する異常発生時にエラーを返す機構を与える。

0.2 プログラミングパラダイム

プログラミングパラダイムとは，プログラムのデザインや実装の方法のモデルである。異なるモデルは異なるテクニックを与える。異なるテクニックは互いに対立するものではなく，多様なテクニックは相互に補い合うものと考えることができる。すべてのプログラミングモデルに共通することは，デザインがプログラミング上の問題の要素に対応する抽象概念に基づくべきであり，実装は再利用可能なモジュールの集まりであるべきだという考え方である。モデル間で異なるのは，抽象概念を構成するものや，モジュールを構成するものである。

すでに確立している手続き型プログラミングの手法は，プログラムを関数の集まり

として構築するモデルに基づいている。手続き型プログラミングのテクニックは，プログラムを構成する関数をどのようにデザインし，構成し，実装するかというガイドラインを提供する。機能を分解していくこのデザイン法では，関数をプログラミング上の問題を解決する抽象演算の役目を果たすものとみなす。ファイルの構成によって，関数をそれぞれのモジュールごとにグループ分けすることができ，構造化プログラミングのテクニックが，関数の実装を読みやすくし，メンテナンスを容易にする。

データ抽象は，手続き指向のプログラミングでは見過されていたデータ構造に焦点を当てたものである。データ構造は，データが実装されている構造よりも，むしろデータの演算によって定義されるべきであるというのがデータ抽象のモデルである。データ抽象のテクニックでは，抽象データ型でデータ構造を覆い隠し，データ構造へのアクセスは，型に用意された一連の演算を通して行われる。データの抽象表現と手続き型プログラミングは，抽象演算という機能面で互いに補足し合い，これらの両者が揃うことではじめて完全なものになる。

オブジェクト指向プログラミングは，抽象データ型から生成されたオブジェクトの集まりでプログラムを構築するモデルに基づいている。オブジェクト指向デザインでは，プログラミング上の問題にかかわる各オブジェクトを表現する型と，これらの型の間の関係を特定する。オブジェクト型に関する演算は，手続き型プログラミングモデルでの関数と同様に，問題を解決する抽象演算である。オブジェクト型は，同じ問題領域のほかの問題を解くために再利用できるモジュールとしての役割を果たす。

プログラミング上の問題を，すべてそれ１つでうまく解決できるようなパラダイムはない。プログラミングは，工学の専門知識を必要とするがまだ科学とはいえない。プログラミングテクニックは，当面の問題の解決にうまく当てはまる方法を考えながら，柔軟に適用される必要がある。やみくもに，現在最も流行しているパラダイムを使ってみても，問題をきちんと調べて注意深い抽象表現を行ったことにはならない。本書の目標の１つは，プログラムのデザインと実装の問題を，批判的かつ柔軟に読者に考えてもらうことである。

0.3　本書の構成

本書の目的は，読者にC++を使ったプログラムのデザインや実装の方法を教示することである。本書では，C++言語の機能を紹介するにあたって，C++の機能がどんなテ

クニックやパラダイムのためにデザインされたかの議論も織り混ぜるという構成をとった。まず，データ型と演算について述べ，次にデータ構造と演算をプログラムにまとめる手段を解説し，最後にメモリ管理やライブラリデザインの話題へと進めていくつもりである。

プログラムは，データに対する演算を順番に記述したものだという考え方がある。これに従って，第1章ではC++のデータ型と演算を紹介する。この言語がCから引き継いだ特徴として，数値を表現する抽象データ型と，対応する算術演算子がある。さらにC++には，言語に新たなデータ型と演算を追加するための利用者定義のクラス型がある。

第2章では，一連の演算を関数にまとめる手続き型プログラミングテクニックについて論じる。機能分解や構造化プログラミングは，デザインに関するパラダイムとして扱われる。ここでは，名前のスコープとリンケージ，関数の値呼び出しとリファレンス呼び出し，関数の多重定義，引数のデフォルト値，テンプレート関数，インライン関数といった言語機能も解説する。

第3章では，本書の以降の章の土台となるC++のクラスの基本機能を取り扱う。データメンバとメンバ関数，コンストラクタとデストラクタ，演算子の多重定義，そして「データの保護」についての解説を行う。

第4章では，データの抽象表現のためのクラスの使用法を論じる。実装を安全でメンテナンスしやすいものにするには，新しいデータ型をどのようなクラスとして開発したらよいかを論じ，新しい型を既存の型システムにまとめるために，型変換をどのように定義したらよいかを示す。

第5章では，クラスの継承を紹介する。クラスの継承とは，既存のクラス型から新しいクラス型を定義する機能である。継承は，既存の抽象データ型を修正したり，関連する抽象型の階層的な集まりを構成したり，関連のない型の性質を結び付けたりするために使われる。クラスの継承は，関数呼び出し時に動的に型を決定するという，柔軟性に富んだ実行時の関数束縛機構も提供する。

第6章では，オブジェクト指向プログラミングのデザインに，データ抽象と継承を応用する。オブジェクトの型のモジュール群としてプログラムをデザインし，構成するためのガイドラインを示し，動的なオブジェクト指向プログラミングのスタイルを説明する。

第7章では，C++の記憶管理機能を紹介する。この機能を使えば，通常の使用のために，あるいは既存のライブラリや利用者定義のクラス型のために，または特定のアプリケーションのために，効率が高く，メンテナンスしやすい記憶管理機構を開発することができ，既存の機構を修正して利用することもできる。

第8章では，再利用可能なソフトウェアモジュールの集まりであるライブラリについて，デザインと使用法を論じる。ライブラリのソースに影響を及ぼすことなく，利用者が拡張したり修正したりできるようなライブラリのデザイン法を示す。

各章の最後に演習を付けた。「†」が付加されているものは，付録Aに解答がある。

私達は，C++の細部を，ソフトウェアデザインあるいはソフトウェア工学のもっと大きな問題に関連付けて解説を行った。私達は，読者がC++をプログラミングツールとして効果的に利用する手段を学ぶだけでなく，プログラムのデザインや実装の過程を新しい手段で考えるための刺激を受けることを期待している。

1

データ型と演算

C++の型システムは，言語定義の組み込み型と利用者定義のクラス型と，組み込み型とクラス型からの派生型から構成される。C++は，組み込み型に対する演算子と型変換を標準で提供している。利用者定義のクラス型にも，組み込み型と同様の演算子と型変換を持たせることができるため，言語定義の型システムを，クラスを使って一貫性を保ちながら，自然に拡張していくことができる。

C++の組み込みのデータ型は，それぞれの型の値を扱う際に使い分けられる演算子の種類に応じて，数値あるいは論理値(ブーリアン)といった抽象概念として解釈されたり，コンピュータの内部表現に従ったビット列として解釈されたりする。算術演算子と関係演算子と論理演算子は，型の数学的な解釈あるいは論理的な解釈の結果を返す。このほかの演算子は，ビット表現によって解釈した結果を返す。

1.1　数値型

C++には，整数と浮動小数点数という2種類の数値型がある。この言語は，整数と浮動小数点数の両方に適用できるように多重定義された算術演算子と，これらの数値型の間の型変換機能を提供している。多重定義演算子とは，同じ記号("+"や"-"など)で，異なる2つ以上の演算の実装を表現している演算子のことである。数値型は，これら算術演算子と一緒に使われる際には，数値の表現として解釈される。

次のプログラムは，数値変数が格納している数値を，算術演算子を使って計算するものである。整数を表現するためにint型の値を使い，実数の表現にはdouble型の浮動小数点数値を使っている。

```
#include <stdio.h>

main() {
/*
    物体の落下後の距離を，最初の10秒間について
    1秒ごとに表示する(単位: メートル)。
*/
    const double g = 9.80;   // 重力加速度
```

```
        for( int t = 1; t <= 10; t++ ) {
            double distance =  g * t * t / 2;
            printf( "\t%2d %7.2f\n", t, distance );
        }
    }
```

このプログラムは，物体の落下後の距離を最初の10秒間について1秒ごとに表示する。

```
     1    4.90
     2   19.60
     3   44.10
     4   78.40
     5  122.50
     6  176.40
     7  240.10
     8  313.60
     9  396.90
    10  490.00
```

距離を計算する式には，double型とint型のオペランドが混在している。結果は，double型の変数distanceに格納されたdouble型の値である。

```
double distance =  g * t * t / 2;
```

この式では，“t”と“2”はどちらもint型であるが，計算はすべて浮動小数点演算で行われる。変数“g”はdouble型であり，乗算や除算は左から右へと評価されるため，各演算子は最低1つのdouble型のオペランドを持つことになる。このため，int型のオペランドは強制的にdouble型に変換される。

数値演算子は，複数の**数値型**に対して多重定義されており，オペランドの型によって，整数演算あるいは浮動小数点演算のどちらを行うかが決定される。したがって，違う順序で演算が行われるように式を変更すれば，いくつかの演算の型も変わることになる。

```
distance = g * ( t * t / 2 );
```

整数のオペランドの集まりを“()”で囲めば，“t * t”の乗算は整数演算で行われ，結果はint型になる。次の2による除算もint型のオペランドに対して適用されるため，int型の結果となる。このとき，tが奇数であれば，結果の小数部は捨てられ，この計算式は浮動小数点数による前回の計算式とは違う値を算出する。除算のオペランド

として浮動小数点定数を使用すれば，除算は強制的に浮動小数点数演算で行われ，前回と同じように結果を正確に得ることができる。

```
distance = g * ( t * t / 2.0 );
```

多重定義演算子にとって，オペランドの型は，演算にあたって演算子のどの実装を選択して実行するかを決める役目を果す。

算術演算子は，何種類かの型のオペランドに処理できるように定義してある。単項算術演算子には，++(インクリメント)，--(デクリメント)，-(負符号)，+(正符号)があり，二項算術演算子には*(乗算)，/(除算)，+(加算)，-(減算)がある。加えて剰余演算子%は，整数のオペランドだけを処理することができる。

インクリメント演算子とデクリメント演算子には，オペランドであるオブジェクトの値自体を変えるという副作用がある。これらの演算子は前置形式か後置形式のいずれかで使用でき，それぞれ異なる結果を返す。前置形式では，式の値はインクリメントないしデクリメントされた後のオブジェクトの値であり，後置形式では，式の値は変更される前のオブジェクトの値である。これら以外の算術演算子はオペランドの値を変えない。

単項演算子は二項演算子より優先順序が高い。乗法二項演算子(*，/，%)は，加法二項演算子(+，-)より優先順序が高い。

整数型には，char，short，int，longがある。これらの型は，この順序で1つ前の型の数値の範囲を最低限表現できるので，いずれの型も1つ前の型よりサイズが大きいか等しいと考えることができる。整数型には，それぞれsignedとunsignedという2つの版がある。宣言では，short，int，longの型指定子は，明示的にunsignedと指定されていなければ，signed版である。異なる整数型のオペランドが式の中で混在している場合，それらは同じ型に変換される。このとき，charとshort型のオペランドは，どの場合でも，いったんint型に変換される。このほかの型は，常に大きいほうのオペランドの型に変換される。どちらかの整数型のオペランドがunsignedであり，もう1つのオペランドの型よりサイズが大きいか等しいならば，オペランドの値はunsigned型にだけ変換される。C++の処理系のいくつかは“long long”型

をサポートしている。この型は大容量の記録装置へのアクセスのオフセット値といった非常に大きい数値を格納するために使われる。

浮動小数点型には，`float`，`double`，“`long double`”がある。これらの型は，この順序で１つ前の型の数値の精度を最低限表現できるので，いずれの型も１つ前の型を部分集合にしていると考えることができる。異なる浮動小数点型のオペランドが算術式の中で混在している場合，サイズの小さいほうのオペランドは大きいほうのオペランドと同じ型に変換される。浮動小数点数と整数のオペランドが混在している場合，整数は浮動小数点数のオペランドと同じ型に変換される。

どの数値型のオブジェクトにも，あらゆる数値を代入することができる。この場合，数値は代入式の左辺の型に変換される。

```
double d;
d = 42;
```

型変換の結果，数値の一部が失われることもある。

```
int i;
i = 3.1415;
char c;
c = 777;
```

この例では，`i`の値は数値3である。数値777はほとんどの場合`char`で表現するには大きすぎるであろうが，代入後の`c`の値が何であるかは，C++の処理系が大きなサイズから小さいサイズへの整数の型変換をどのように行うかに依存している。

型変換演算子(いわゆるキャスト)によって，明示的に型変換を行うこともできる。次の例では，`count`の`int`型の値は明示的に`double`型に変換される。

```
int count = 1066, total = 1337;
double ratio;

ratio = double( count ) / total;
```

この例では，関数書式のキャストを使っている。

```
double( count )
```

同じキャスト演算を行うもう 1 つの書式を示す。

```
( double )count
```

型変換の式が単純であれば，通常は関数書式のほうがわかりやすいだろう。

プログラム中で，数値を数値定数として直接表現することができる。浮動小数点定数は，小数部と指数部で表現される。

```
9.80
0.98e1
98e-1
```

これらの定数はすべて，double 型の同じ値を表現する。このほかの浮動小数点型では，“long double” には後置文字 “L” または “l”，float には後置文字 “F” または “f” を付加することによって表現する。

```
0.98e1L
9.80f
```

整数定数は，値がint に格納するのに大きすぎず，しかも後置文字で型が示されていない場合は，int 型である。またint 型で表現するには大きすぎる定数は，long 型で扱うことができる。定数の後置文字L またはl はlong 型を示し，U またはu はunsigned 版を示している。定数の後置文字は，次の例のように複数指定してもよい。

```
1642UL
```

最初の桁が “0” で始まるものは 8 進定数であることを示している。

```
0777
```

先頭の “0x” または “0X” は 16 進定数であることを示す。16 進定数中の文字は大文字でも小文字でもよい。

```
0x1ff
0X1fF
```

const と初期値によって，数値を記号的に表現することもできる。

```
const double g = 9.80;
```

この場合の“const”はgの値が変更できないことを示しているので，識別子gは常に初期値9.80を表現する。

さらにenumの要素として，整数値を記号的に表現することもできる。初期値が特に設定されていなければ，enumの要素は連続する整数の型の値をとり，最初の要素の値は0である。

```
enum { MON, TUES, WED, THUR, FRI, SAT, SUN };
```

ここでの“FRI”の値は4である。enumの要素を明確に初期設定することもできるが，この場合も，リスト上にある初期設定されていない要素は，リストの順序で1つ前の要素値より1だけ大きい値を持つ。

```
enum { MON = 1, TUES, WED, THUR, FRI, SAT = -1, SUN };
```

ここでのFRIの値は5，SUNの値は0である。

enumに，整数の型の名前としてタグ名を与えて宣言してもよい。

```
enum Day { MON, TUES, WED, THUR, FRI, SAT, SUN };
```

enum型として宣言された変数に代入してよいのは，同じ型の値だけである。

```
Day aday;

aday = FRI;    // OK: FRIはDay型の値
aday = 18;     // エラー！ Dayにintを代入しようとした
aday = 4;      // エラー！ FRIの値は4であるが，4はintである
Day aday;
```

しかし，式の中では，enum型の値はint型に自動的に変換される。int型はenum型に変換されてもよいが，正しい結果を得るには，整数値はenumの範囲に収めるべきである。

```
aday = Day(4);   // OK: FRIの値は4である
aday = Day(18);  // 未定義動作！ プログラムエラー
```

1.2　文字型

C++には，charとwchar_tという2つの文字型がある。これらは整数の型であり，プログラム中で，所定の文字集合中の文字に対応するコードを格納するのに使われる。

C++ではラテン系の文字集合が使われ，この文字集合はcharデータ型に格納することのできるコード値で表現される。このC++の文字集合は，英語系のテキストを表現するのには十分である。また，ヨーロッパ系言語あるいは文字数のわずかであるアルファベットも，単一バイト(char型)エンコードで表現される。

しかし，日本語，中国語，韓国語の文字集合の要素は，1バイトあるいは複数バイトの列にエンコードされて表現される。単一の文字集合であっても，要素ごとに違うバイト数で表現されることもある。C++では，こうした文字集合の多バイトエンコードを独立したwchar_t(いわゆるワイド文字)型の値に変換して扱うことができる。wchar_t型では，すべての文字は同じバイト数で表現され，プログラム中での操作も容易である。

C++のコードは，英語を基本にしているが，注釈，ヘッダファイル名，文字定数には，地域依存の文字集合中の文字を使うことができる。たとえば，日本語環境では，以下のコードが許される。

```
#include "日本語.h"
// これは日本語の注釈です。
```

文字定数(「'」で囲まれた文字またはエスケープシーケンス)は，文字の値を表現するための便法を与える。

```
char w = 'w';
char newline = '\n';
char tab = '\t';
char null = '\0';
char bell = '\007';
```

ワイド文字定数は，前置文字Lで示される。

```
wchar_t wc = L'あ';
```

エスケープシーケンスを使って，ワイド文字の値を示すこともできる。

```
wchar_t wchar_null = L'\0';
```

\で始まり，次に3つの8進数字が続くエスケープシーケンスは，文字記号の代わりに8進数で文字の値を与える。同様に，\xで始まるエスケープシーケンスは，16進数で文字の値を与える。

C++の文字コードの値は，たとえ“signed char”に格納されていたとしても，常に正の値を取る。文字の値は，式の中で整数のオペランドとして使うこともできる。

```
char digit = '9';
int value = digit - '0';
```

文字の値の間の算術演算は，通常，標準文字エンコードを仮定して行われる。この例は，ASCIIコード値を仮定している。

概して，C++で使われるワイド文字コード値は，テキストファイルに格納するときに使われる日本語，中国語，韓国語の文字の多バイトエンコードとは異なることが多い。このため，ワイド文字と多バイト文字間の変換を行うために，mbtowc，wctombといった標準関数が用意されている。これらの関数は，ヘッダファイルstdlib.hで宣言されている。これらの関数の動作は，地域依存の環境と処理系に依存する。

1.3　関係演算子と論理演算子

C++には，論理値(ブーリアン)はなく[1]，代わりにスカラー型，たとえば整数などを論理値として扱う。この場合，値が0の場合に偽(false)を表し，0以外の値はすべて真(true)を表す。1.6節で述べるポインタもスカラー型である。関係演算子は値を比較するために使われ，真偽の結果を1か0の整数値として値で返す。論理演算子は，真または偽と解釈されるスカラー型のオペランドに作用し，関係演算子同様に，結果を1か0の値で返す。

関係演算子は，結果として1か0のint型の値を返す。落下距離を計算する1.1節の例では，繰り返しを制御する式に関係演算子を使った。

```
for( int t = 1; t <= 10; t++ ) {
    // etc.
}
```

1) 訳注：新しい標準規格案では，論理値を表現するためにbool型が導入された。付録B.1参照。

制御条件を強調するために，このforの繰り返しをwhileを使って等価な繰り返し処理に書き直してみよう。

```
int t = 1;
while( t <= 10 ) {
   // etc.
   t++;
}
```

繰り返し本体は，条件式“t <= 10”が0と評価されるまで実行され続ける。この式は，tが10より小さいか等しい場合に1と評価されるので，tの値が1から10までの間，繰り返しが実行される。

関係演算子には，“<”(小さい)，“>”(大きい)，“<=”(小さいか等しい)，“>=”(大きいか等しい)がある。また，等値演算子には，“==”(等しい)と“!=”(等しくない)がある。これらの演算子は，整数，浮動小数点数あるいはポインタのオペランドに対して多重定義されているが，どの場合でも1か0のint型の値を結果として返す。

0以外の値はすべて真を表すので，1か0の値を返す式のほかにも，あらゆる式を条件式として使うことができる。たとえば，

```
int t = 11;
while( --t ) {
   // etc.
}
```

この繰り返し本体は，tの値が10から1までの範囲である間，実行される。

繰り返しの制御では，一般的には明示的に比較したほうがよい。たとえば，

```
while( --t > 0 ) {
   // etc.
}
```

この例では，tが最初から0あるいは負数である場合，繰り返しは実行されない。

論理演算子には，“&&”(論理積)，“||”(論理和)，“!”(論理否定)がある。単項演算子“!”は，オペランドが非0であれば0，0であれば1を式の結果として返す。次の例は，変数の値が0の場合だけ値を新たに設定するものである。

```
if( !p ) // i.e. if( p == 0 )
   p = get_a_val();
```

“||”を使った式は，オペランドがどちらも0ならば0，さもなければ1を返す。次の例は，xの値が0から10までの範囲に収まっていない場合だけ関数errorを呼び出す。

```
if( x < 0 || x > 10 )
    error("value outside range");
```

“&&”を使った式は，オペランドがどちらも非0の値であれば1，さもなければ0を返す。次の例は，前の例と同じ条件を違う論理式を使って調べている。

```
if( !( x >= 0 && x <= 10 ) )
    error("value outside range");
```

この式よりも前の式のほうがわかりやすいため，できるだけ前者の表現を使用したほうがいいだろう。

式の結果を決定するにあたって不要であれば，&&あるいは||の第2オペランドは無視される。“||”式の第1オペランドが非0であれば，結果は第2オペランドの値に何であろうと1である。同様に，“&&”式の第1オペランドが0であれば，第2オペランドは評価されず，式の結果は0である。次に示す例では，bの値が0であれば，等値式は評価されず，除算の式も同様に無視される。

```
if( b && a/b == c ) {
    // etc.
}
```

ここで&&を使用してbが**非0**であることを調べているのは，0による除算を避けるためである。&&あるいは||の第2オペランドを評価した際の副作用を当てにするのは誤りである。次の例では，aが非0であれば，関数fは呼び出されない。

```
if( a || f() ) {
    // f()の副作用を当てにしたコードはエラー！
}
```

1.4 非抽象演算

C++には，型の抽象的な解釈を飛び越して，機械表現を直接扱うことを許す演算子がいくつかある。

整数型は，コンピュータの種類ごとに異なる長さのビット列として実装される。整数は，算術演算子，関係演算子あるいは論理演算子と一緒に使われる際には，数値あるいは真偽値として抽象的に解釈されるので，ビット表現の細部は無視してしまってもよい。しかし，たまにビット表現を扱いたいと思うこともある。

以下の例は，Peter Weinberger 氏が考案した文字列ハッシュ関数である。ハッシュ関数は，文字列をキーとした情報検索を行うために，与えられた文字列の内容に従って，情報の記憶位置を決定する数値を算出する。仮引数は，数値に変換して使われる文字列へのポインタである。変数hashは，最後に素数を格納した変数primeとの剰余を求める際に初めて数値として扱われるまでは，32 ビットのビット列として操作される。

```
int
hashpjw( char *s ) {
    const prime = 211;
    unsigned hash = 0, g;

    for( char *p = s; *p ; p++ ) {
        hash = ( hash << 4 ) + *p;
        // int の大きさを 32 ビットに仮定する
        if( g = hash & 0xf0000000 ) {
            hash ^= g >> 24;
            hash ^= g;
        }
    }
    return hash % prime;
}
```

この例では，“<<”(左シフト)，“>>”(右シフト)，“&”(ビット毎の論理積)，“^”(ビット毎の排他的論理和)というビット毎の演算子が使われている。このほかにも，“|”(ビット毎の論理和)，単項演算子 “~”(ビット毎の補数)が利用できる。

例では，演算子^を代入演算子形式(^=)で使っている。

```
hash ^= g;
```

は次の式と等価である。

```
hash = hash ^ g;
```

二項演算子 *, /, %, +, -, <<, >>, &, ^, | は，すべて同様にして代入演算子と結合することができる。

演算子“sizeof”は，指定された型の表現に使われるバイト数を与える。たとえば，

```
sizeof( int )
```

の結果は，int 型を表現するのに要するバイト数を与える。

```
sizeof( char )
```

の値は常に1である。また，sizeof に型の代わりに式をオペランドとして与えると，式の型のサイズを返す。この演算子の一番の使い道は，オブジェクトを動的に生成する際に要する領域のサイズを決定することであった。しかし C++では，記憶管理演算子 new と delete の内部に sizeof の機能が組み込まれているので，これらの記憶管理演算子を使うのであれば，演算子“sizeof”を使って型表現のサイズを知る必要はほとんどない。

1.5 利用者定義の型

C++の利用者は，C++に標準で用意されている基本の型システムを拡張する目的で，クラス型を定義することができる。利用者定義のクラス型は，演算子関数と型変換を定義することによって，ほかの型と一緒に使うことができる。

以下に示したのは，複素インピーダンスの公式 $Z = R + j\omega L + 1/(j\omega C)$ とオームの電圧の公式 $V = ZI$ を使って，誘導インダクタンス，抵抗，コンデンサ容量から成る交流電気回路の電圧を計算するプログラムである。この交流回路で，電圧，電流，インピーダンスは複素数の実数部と虚数部で表される 2 つの要素を持つ。しかし C++には，言語定義の複素数の型が用意されていないので，このプログラムでは，複素数の値を表現するのに利用者定義のクラス型を使っている。

```
#include "complex.h"

main() {
/*
    交流回路の電圧を計算する
*/
    const complex j( 0, 1 );    // 虚数単位
    const double pi = 3.1415926535897931;

    double
        L = .03,      // インダクタンス(ヘンリー)
        R = 5000,     // 抵抗(オーム)
        C = .02,      // 容量(ファラッド)
        freq = 60,    // 周波数(ヘルツ)
        omega = 2 * pi * freq;
                      // 角周波数(ラジアン/秒)
    complex
        I = 12,       // 電流
        Z,            // インピーダンス
        V;            // 電圧

    Z = R + j * omega * L + 1/( j * omega * C );
    V = Z * I;
    V.print();
}
```

このプログラムの出力は次のようになる。

```
( 60000.00, 134.13 )
```

ヘッダファイルcomplex.h では，複素数の数学的な概念を実装したクラス型complexの定義を行っている。以下に，クラス型complexの例として要約版を示す。

```
class complex {
    double re, im;
  public:
    complex( double r = 0, double i = 0 )
        { re = r; im = i; }
    void print();
    friend complex operator +( complex, complex );
    friend complex operator *( complex, complex );
    friend complex operator /( complex, complex );
};
```

このクラス定義は，クラスのメンバ(データメンバと関数メンバ)の宣言とfriend(フレンド)関数の宣言から成る。フレンド関数は普通の関数であるが，これらの関数にはこ

のクラスのメンバへのアクセス権が特例的に与えられる。public(公開)部を示すラベル “public” の後に宣言されたメンバには制限なしにアクセスできるが，private(非公開)メンバreとimには，メンバ関数とフレンド関数だけがアクセスできる。このcomplexは，public部で宣言されている関数を通してのみ利用できるようにするために，内部表現であるデータメンバはすべて，クラスのprivate部に隠されている。

クラス型名と同じ名前のメンバ関数は**コンストラクタ**と呼ばれる。このコンストラクタは，complex型のオブジェクトを生成して初期設定するために，あるいはほかの型の値をこのクラス型の値に変換するために使用される。この例のコンストラクタには，初期値が与えられなかった場合に使われるデフォルトの引数値が宣言されている。このため，実引数の個数が0個，1個，2個のいずれであっても，このコンストラクタを呼び出すことができ，必要であればデフォルトの引数値が使われる。

```
const complex j( 0, 1 );
```

この変数宣言では，コンストラクタの引数を2つとも初期設定している。つまり，jの実数部と虚数部は，それぞれ0と1である。

```
complex I = 12;
```

という宣言は，

```
complex I( 12, 0 );
```

と同じ意味である。これはデフォルトの引数値が，Iの虚数部を設定する引数の値を0にするからである。先程のプログラムで示したZとVでは，宣言で初期値を与えていないため，コンストラクタの引数はどちらもデフォルト値で初期設定される。

クラスcomplexで宣言されている(コンストラクタ以外の)関数は，すべてクラスの外側で定義してある。

```
void
complex::print() {
    printf("( %5.2f, %5.2f )\n", re, im );
}
```

```
complex
operator +( complex a1, complex a2 ) {
   return complex( a1.re + a2.re, a1.im + a2.im );
}

complex
operator *( complex a1, complex a2 )
{
   return complex(a1.re * a2.re - a1.im * a2.im,
            a1.re * a2.im + a1.im * a2.re);
}

complex
operator /( complex a1, complex a2 )
{
   double r = a2.re; /* (r,i) */
   double i = a2.im;
   double ti;        /* (tr,ti) */
   double tr;

   tr = r < 0? -r : r;
   ti = i < 0? -i : i;
   if( tr <= ti ) {
      ti = r/i;
      tr = i * (1 + ti*ti);
      r = a1.re;
      i = a1.im;
   }
   else {
      ti = -i/r;
      tr = r * (1 + ti*ti);
      r = -a1.im;
      i = a1.re;
   }
   return complex( (r*ti + i)/tr, (i*ti - r)/tr );
}
```

先程の例では，メンバ関数printはVに格納された演算結果の値を表示するために使われた。

```
V.print();
```

メンバ関数printは，complexクラスのすべてのメンバに制限なくアクセスでき，呼び出し対象のcomplexオブジェクトのメンバreとimの内容を特定の書式で表示する。

演算子を定義する関数(以下，**演算子関数**と呼ぶ)は，クラスcomplexでは，すべてフレンド関数として宣言されている。フレンド関数はクラスのメンバではないが，メンバ関数と同様に，complexオブジェクトのprivateメンバをアクセスすることができる。これらの演算子関数はcomplexの値に対する数値演算を実装しているので，中置形式の式でcomplex型のオペランドを使うことができる。

```
Z = R + j * omega * L + 1/( j * omega * C );
```

このインピーダンスを計算する式には，int，double，complexといった各種の型のオペランドが混在している。このようにintまたはdoubleのオペランドがcomplex型のオペランドと同居している場合，自動的にコンストラクタが呼び出されてオペランドがcomplex型に変換され，変換後の値を引数として演算子関数が呼び出される。組み込みの型の間の言語定義の型変換によって，int型の値はコンストラクタの正しい引数型であるdouble型に変換されるため，intあるいはdoubleの値を1つのコンストラクタでcomplex型に変換することができる。コンストラクタcomplexが型変換のために使われるとき，第2引数はデフォルト値で初期設定される。

クラスの内部表現の隠蔽と利用者定義の演算子と型変換によって，クラスcomplexは，言語定義の数値型と自然に混在できる抽象的な数値型となる。

1.6 ポインタと配列

ポインタと配列はほかの型から導き出される型として存在する。ポインタ型はほかの型のオブジェクトのアドレスを表現する。ポインタ型は，動的にオブジェクトを配置することでデータ構造に柔軟性を持たせるために使われるほか，ポインタ算術演算子によって配列の要素をアクセスするためにも使用される。配列型は，特定の型の要素の列を表現するものであり，集合データ構造として多くの使い道がある。文字の列を文字列として扱うことは，配列の標準的な使い道の1つである。

ポインタ型は，宣言で，型情報に加えて型修飾子“*”を使うことで示される。同じ記号“*”はポインタを経由した間接参照演算子としても使われ，演算子はポインタが

指す先のオブジェクトを返す。参照結果は，オブジェクトの値の表現として，あるいは代入式の左辺値として使用される。

```
int *p;          // pはintへのポインタ
int i = 33;
p = &i;          // pにiへのポインタを設定
*p = *p + 1;     // iを34に設定
```

演算子“&”は，オペランドであるオブジェクトのアドレスを返す。アドレス値の型は,「ある型のオブジェクトへのポインタ型」である。この例では，式&iの型は「intへのポインタ型」である。

演算子newは，オペランドが示す型のオブジェクトをフリーストア領域に動的に生成し，新しいオブジェクトへのポインタを返す。演算子newが生成したオブジェクトは，演算子deleteによって破棄することができる。演算子deleteは，オペランドとして，破棄しようとするオブジェクトへのポインタを取る。

```
int *p = 0;

if( !p )
   p = new int;

delete p;
p = 0;
```

ここで示したコードでは，最初にポインタを宣言し，0に初期設定している。0は特殊なヌルポインタ値であり，無効アドレスを意味する。次に，ポインタが正しい値に設定されているかどうかを検査し，無効アドレスであれば，新たに生成したオブジェクトのアドレスを設定する。最後に，オブジェクトは削除され，ポインタの値もヌルに戻される。ポインタの設定，フラグとしてのヌルポインタの使用，演算子newと演算子deleteによるオブジェクトの生成と破棄は，C++での動的データ構造を構築するための基礎である。

代入と初期設定の一部の限られた場合を除けば，通常，ポインタ型の間で自動的に型変換されることはない。ただしC++には，あらゆるポインタ型の値を格納することができる特殊なポインタ型“void *”が存在する。この“void *”は「汎用ポインタ」であると考えることができる。したがって，“void *”型への値の代入あるいは

“void *” 型の初期設定時には，ほかのポインタ型はすべて “void *” 型に自動的に変換される。

```
int *ip;
void *vp = ip;
double *dp;
vp = dp;
ip = (int *)vp; // 悪い
```

この “void *” 型のポインタは，プログラマがキャストで明示的に指定して，ほかの型へのポインタ型に変換してやる必要がある。しかし，こういった明示的な型変換を行うと，ポインタが「指す先の」オブジェクトを一貫した手法で解釈することを保証する型検査を無視することになるので，危険である。なお，関連するクラス型へのポインタ型の間では自動的に型変換が行われる。これについては第 5 章で解説する。

同じ型の要素を連続して配置したものが配列である。配列の各要素のアドレスは，ほかの要素のアドレスから，ポインタ型のために多重定義された算術演算子を使って計算することができる。ポインタ演算の使用法を紹介するために，文字列の要素を終わりまで走査する繰り返し処理を示そう。

```
for( char *p = s; *p ; p++ ) {
   // etc.
```

ここで，“s” はchar 型の配列であり，文字列を表現している。ポインタ “p” は配列中の最初の要素を指すように初期設定され，値 0 の要素が現れるまで，連続して次の要素を指すようにインクリメントされる。通常，文字列は値 0 の要素で終了するという約束がある。

インクリメント “++” とデクリメント “--” は，ポインタ型のオペランドに対しても作用する。この場合，ポインタは配列要素のアドレスと仮定される。インクリメントは，配列の直後の要素を指すようにポインタを変更し，デクリメントは直前の要素を指すように変更する。

二項演算子の加算 “+” と減算 “-” は，1 つのオペランドがポインタ型，もう 1 つのオペランドが整数型である場合に作用するように定義されている。ただし減算では，整数は第 2 オペランドのほうでなければならない。この場合も，ポインタは配列要素のアドレスであると仮定される。つまり演算の結果として得られるものは，配列のほか

の要素のアドレスである。たとえば，次の加算は，pがもともと指していた要素から数えて３番目の要素を指すようにpを変更する。

```
p += 3;
```

次に示す減算は，もとの要素の２つ手前の要素を指すようにpを設定する。

```
p -= 2;
```

減算は，同じ配列中の要素のアドレスである２つのポインタ型のオペランドに作用するようにも定義してある。結果は，ポインタが指す配列の位置の間の要素の数を示す整数値である。

配列型は型修飾子“[ ]”によって指示される。配列宣言の角括弧“[ ]”の中には，配列の要素数を指定する整数の定数式を与える。たとえば，

```
char buffer[100];
```

は100個のchar型の要素を持つ配列を宣言する。ここで“buffer”という名前は，配列内の最初の要素のアドレスを示す。この名前は，ポインタ算術演算子とともに，配列の要素をアクセスするために使われる。次の繰り返し処理は，配列bufferの各要素の値を0にするものである。

```
for( int i = 0; i < 100; i++ )
    *( buffer + i ) = 0;
```

配列要素をアクセスするのにポインタ式を使うことができる。次の例では，pは最初に配列bufferの第１要素のアドレスに設定され，bufferの先頭の20個の要素を，nameという，もう１つの配列にコピーするために使用されている。

```
char name[21];

p = buffer;
for( i = 0; i < 20; i++ )
    name[i] = *p++;
name[i] = '\0'; // 文字列の終わりを設定する。
```

添字付け演算子“[ ]”は，配列要素をアクセスするポインタ式の短縮表記を提供する。添字付け演算子による，

```
name[i]
```

という表記は，

```
*(name + i)
```

と同じ意味である。

添字付け演算子は，配列名に対してだけでなく，あらゆるポインタ名に対して使ってもよい。次の式はpの1つ前の位置をヌルに設定する。

```
p[-1] = '\0';
```

C++での文字列の基本表現は，値が0の要素で終了するchar型の配列中の連続する文字の値である。ワイド文字列も同様にwchar_t型の配列で表現される。文字列定数は，こういった配列の表記法を提供する。文字列定数は「"」で囲まれた連続する文字またはエスケープシーケンスである。

```
"This is a string literal\n"
```

ワイド文字列定数は，先頭のLで示される。

```
wchar_t *wcs = L"漢字ABC\n";
```

文字列定数の値は，文字型(char型あるいはwchar_t型)の配列の先頭要素へのポインタである。配列のサイズは「"」で囲まれた文字の数より1つ大きく，配列の要素は対応する文字の値を持ち，最後の要素に値0が入る。

1.7 リファレンス

リファレンス型はオブジェクトに別名を付けるための機能であり，ほとんどの場合，関数の値呼び出しの代わりに**リファレンス呼び出し**を実現するために関数の引数型として使われる。

リファレンス型は，ポインタ修飾子と同じように，宣言において，修飾子“&”を使って示される。この際，常に，リファレンスが指し示す先のオブジェクトを初期設定子として与える必要がある。初期設定されたリファレンスによるアクセスは，もとのオブ

ジェクトを直接使った場合とまったく同じ結果を与える。ここでは，リファレンス型がどのように働くかを説明するために，普通の宣言の例から示してみよう。

リファレンスの初期設定では，リファレンスと同じ型のオブジェクトの名前を初期設定子として与える必要がある。

```
int i;
int &ir = i;
```

ここでは，“i”の別名として“ir”を定めている。したがって，irの参照と代入は，iの参照と代入と同じ結果を与える。

```
ir = 3;        // iに値3が代入される
int j;
int *ip;
j = i * ir; // jに値9が代入される
ip = &ir;   // ipにはiのアドレスが代入される
```

いったんリファレンスをオブジェクトの別名として初期設定すれば，後で変更することは不可能である。また，現在のC++の処理系は，リファレンスの型と別名を作成しようとする初期設定子の型が違う場合にはエラーとする。

古いC++では，リファレンスの初期設定子の型が不正であれば正しい型の無名のオブジェクトが生成され，このオブジェクトの別名としてリファレンスが初期設定されていた。以下に，古いC++での動作を紹介しよう。古いC++では，間違った型の初期設定子は型変換され，無名のオブジェクトに値を設定するために使われていた。

```
double d;
int &ir = d;    // 無名のint型オブジェクトが作成される
ir = 3.0;       // dは変更されない！
```

無名のオブジェクトは，初期設定子がオブジェクトでない場合にも作成されていた。

```
int &ir = 3;    // 無名のint型オブジェクトが作成され，3が代入される
```

リファレンスの初期設定時に無名のオブジェクトが作成されてしまうと，プログラムのエラーを発見するのが難しいことが多い。このため，こういった初期設定を許していた昔のC++の処理系であっても，ほとんどのものはこれに対して警告を発していた。

このリファレンス型の使い道としては，先にも書いたように，リファレンス呼び出しを実現するために，関数の仮引数の型として使われることがほとんどである。リファ

レンス呼び出しでは，関数は実引数の値を直接変更することができる。この場合，リファレンスは関数の中での実引数の別名を定めるために使用され，読み書きあるいは書き込み専用の引数を実現する。

```
void input( int &, int &, int & );
int a, b, c;

input( a, b, c ); // 実引数に値を設定する
```

古いC++のように，引数型の変換の結果，リファレンス型仮引数が無名の一時オブジェクトで初期設定されてしまった場合，関数はリファレンス型仮引数に値を設定するが，この仮引数はプログラムのほかのところからはアクセスできないという結果をまねく。

```
double d;
input( a, b, d ); // エラー！  dをinput関数で変更できない
```

この理由から，現在のC++の仕様では，正しい型と名前を持ったオブジェクト以外のもので，リファレンスを初期設定することはエラーである。ポインタ型引数を使って，関数の外部のオブジェクトを変更することもできるが，これを行うには，実引数と仮引数を操作するのにアドレスとポインタの演算が必要である。別名を定めるリファレンス型引数は，ポインタ型引数の代用品として便利に使うことができる。

さらに，リファレンス型引数を使えば仮引数を初期設定する際のオーバヘッドを避けることができる。特に，すこし大きすぎるデータ構造を持つクラス型を引数として渡したいときに，リファレンス型引数を使うことは，実行効率を考えると非常に重要である。

```
class List {
    // 100個のint型要素を持つリスト
    int size;
    int elements[100];
   public:
    int &item( int );
    // ...
} list;

void output( List & );   // リストの値をコピーしない
```

この例の関数outputでは，List型の実引数値はコピーされず直接使われ，配列要

素を複製する際のオーバヘッドは回避される。

関数の戻り型をリファレンスにすることもできる。これを使えば，関数を代入の左辺に使うこともできる。先の例に示したメンバ関数List::item()はリストの要素へのリファレンスを返す。

```
int &
List::item( int ix ) {
    // ... ixが範囲中かどうか検査する
    return elements[ix];
};
```

この関数は1つで，リストの要素を参照するのにも，要素へ値を設定するのにも使うことができる。

```
int i = list.item(3);      // 要素値を取得する
list.item(12) = 6;         // リファレンス経由で要素に値を設定する
```

1.8 constによる限定型

型限定子“const”は，オブジェクトの値を直接にも，またポインタを通しても，変更できないということを示す。すでに述べたが，この性質によって，constオブジェクトは，初期値の記号表現として使うことができる。

リファレンス型引数のうち，実引数の内容を変更するためではなく，実引数のコピーを防ぐことを目的とするものには，値を変更できないように，引数をconstとして宣言することができる。“const”限定子は，関数中で実引数の値が変更されることを阻止する。

constで限定されたリファレンス型引数は，型変換で作成された無名のオブジェクトで初期設定されることもできる。つまり，リファレンス型引数であってもconstで限定されていれば，通常の非リファレンス引数への型変換と同じように，実引数をリファレンス仮引数にマッチさせるために型変換が行われてもよいということである。たとえば，complex型の演算子関数を，オーバヘッドを避けるためにcomplex型の引数はコピーされないが，ほかの数値型の引数は型変換とコピーが行われるように変更することができる。

```
class complex {
    // etc.
    friend complex
    operator *( const complex &, const complex & );
};

complex V, Z;
// etc.

V = Z * 12;
```

“Z * 12”という式は，12で初期設定された無名のcomplexオブジェクトとZを引数とするクラスcomplexのフレンド関数“operator *”の呼び出しである。このように無名のオブジェクトによって，constで限定されたリファレンス型引数が初期設定されるため，型の混在した式でも型変換を妨げることなく引数渡しのオーバヘッドを軽減することができる。

ポインタ型の宣言でのconstによる限定は，“const”の位置によって，ポインタ自体が変更できないのか，ポインタが指すオブジェクトが変更できないのかを示す。ポインタ修飾子の前に配置されたconst限定子は，ポインタが指す先のオブジェクトがconstであることを示す。

```
const char *p = buffer;     // “const char”へのポインタ
p = name;                   // OK
*p = 'x';                   // エラー！
```

ポインタ修飾子の後ろに配置されたconst限定子は，ポインタ自身が変更できないことを示す。

```
char * const p = buffer;    // charへのconstポインタ
*p = 'X';                   // OK
p = name;                   // エラー！
```

constオブジェクトのアドレスを非constオブジェクトへのポインタに代入することはできない。この理由は，こういった代入は，ポインタを通してconstオブジェクトを変更することを許してしまうからである。

```
const char space = ' ';
const char *p = &space;     // OK
char *q = &space;           // エラー！
```

演　習

演習 1-1　char，short，int，long 型のオブジェクトを表現するのに何ビットが必要かを表示する C++プログラムを書け。

演習 1-2†　与えられた宣言が，

```
int &a = 12;
int *b;
int *&c = b;
int *d[5];
```

であるとき，次の式の型は何であるか。

```
a
b
*b
c
*c
d[2]
*d
**d
c[-2]
c-2
*(c-2)
&c
```

演習 1-3†　与えられた宣言が，

```
char c;
const char cc = 'a';
char *pc;
const char *pcc;
char *const cpc = &c;
const char *const cpcc = &cc;
char *const *pcpc;
```

であるとき，次の代入のどれが正しくどれが誤りだろうか。また，理由も述べよ。

```
c = cc;
cc = c;
pcc = &c;
pcc = &cc;
pc = &c;
```

```
pc = &cc;
pc = pcc;
pc = cpc;
pc = cpcc;
cpc = pc;
*cpc = *pc;
pc = *pcpc;
**pcpc = *pc;
*pc = **pcpc;
```

演習 1-4　以下の各宣言を文で表現せよ。

```
int **a;
int *&b = 0;
int **&c = 0;
int * const d = 0;
int const *e;
int const *const *f;
int * const* *const g = 0;
int (*h)(int *);
int *(*j)();
int (*&k)() = 0;
int (*l);
int (*m)[12];
int (*n[12]);
int *o[12];
int (*p[12])();
int (**q[12])();
int (*(*r)[12])();
int *&(**(*s[3])[12])();
int (*t(int))(int *);
```

演習 1-5　演習 1-4 の宣言を typedef を使ってわかりやすくせよ。

演習 1-6　整数定数 12 と，以下の宣言で定義された整数の const オブジェクトの違いを述べよ。

```
const int i = 12;
```

演習 1-7　次の各関数の動作を説明せよ。

```
void
swap1( int *a, int *b ) {
   int t = *a;
   *a = *b;
   *b = t;
}

void
swap2( int &a, int &b ) {
   int t = a;
   a = b;
   b = t;
}

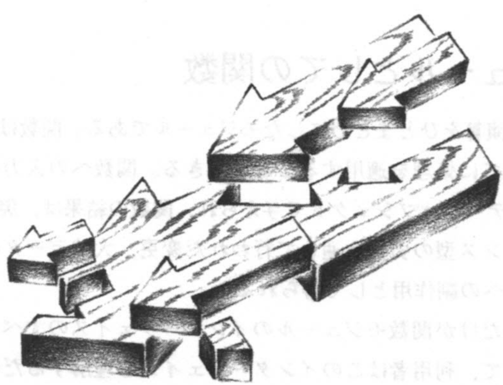
void
flop( int a, int b ) {
   int t = a;
   a = b;
   b = t;
}
```

演習 1-8　所定の定数式が指定された値でない場合に，コンパイル時エラーを起こす機構を考案せよ。また，こうしたテクニックの使い道には何があるだろうか。

演習 1-9　リファレンス型仮引数を使って，詳しいエラー情報を関数の呼び出し側にどのように与えることができるかを示せ。

2

手続き型プログラミング



プログラムは，問題を解決するためのアルゴリズム，いわゆる手続きを実装したもので，これはデータ構造に対する順次の演算(一連の演算)である。この際の問題点のほとんどは，手続きが長く理解しにくいものになってしまい，この大きさと複雑さをうまく管理する手段で使って対処しなければ，プログラムの実装は難しく，メンテナンスの費用も高騰してしまうしまうことにある。手続き型プログラミングテクニックは，この問題に対して，プログラムを分割して構造的に表現する手段を与え，プログラムの作成，理解，メンテナンスを容易にしている。ただし，このテクニックはプログラム中の順次の演算を構成する手段のみに注目していて，演算対象としてのデータ構造はほとんど考慮されていない。

C++でのプログラムの実行は，(静的データ構造の初期設定のあとで)main 関数が実行されることで始まる。このmain 関数にプログラム中のすべての演算が書かれていることはまずなく，いくつかの演算は通常ほかの関数にまかせられる。またいくつかの演算は，第 3 章に示す機能を使ってデータ構造に組み込まれることもある。この章では，やさしい構造のデータ型だけを使って，C++で関数をデザインしたり構成したりする際に，手続き型プログラミングテクニックをどのように利用するかを説明する。

2.1　モジュールとしての関数

関数は，順次の演算をひとまとめにしたモジュールである。関数は，引数を持ち，正しい型の引数すべてに処理を適用することができる。関数への入力は，引数と，関数中で使われる大域データオブジェクトで与えられ，関数の結果は，戻り値，ポインタ型あるいはリファレンス型の引数を通して行われた変更，大域データの変更，あるいはプログラムの外部への副作用として得られる。

この入力と結果だけが関数モジュールのインターフェイスのすべてであり，関数を使用するにあたって，利用者はこのインターフェイスを理解するだけでよい。関数を実装している実際の順次の演算は，細部として隠され，関数自体をただ 1 つの抽象演算と考えることができる。プログラム中に，ある関数をデザインして実装することは，

所定の問題を解決するための専用の**演算**を1つ作成することであると考えられる。

プログラムは，問題を正しく解決するという技術目標に加え，利潤を生み出すという経済目標も達成する必要がある。関数のモジュール性を利用することで，関数をよりわかりやすく，再利用しやすくすることができ，この結果として，プログラムの実装とメンテナンスの費用を削減することができる。

関数をわかりやすくするには，問題解決の抽象演算に対応させて関数をデザインすればよい。この際に，関数(とプログラム中での関数の呼び出し)は，実装の細部ではなく，問題に特有の表現と考えることができる。たとえば，未ソートのリストを入力とし，要素を昇順にソートする関数は，「リストをソートする」という抽象演算と考えることができる。この抽象演算と同じものは，ほかの多くの問題の解法にも使うことができるだろう。したがって，この抽象演算の実装としての関数が，特定のプログラムの文脈から独立するようにデザインしてあれば，ほかの多くのプログラムで，この同じ関数を再び使うことができるわけである。

関数のうち，大域データオブジェクトを使わず，入力として引数だけを持つものは，プログラム中の特定のオブジェクトだけを処理するのではなく，正しい型を持つ引数すべてを処理することができる。したがって，この関数を，元のプログラムの中だけでなく，同じデータ構造の型を使うほかのプログラムの中で，別の入力に対しても再利用することができる。関数のインターフェイスは関数宣言から知ることができ，関数のデータオブジェクトは関数内部の変数宣言から理解できる。引数入力と局所オブジェクトだけを使った関数は，再利用が容易であることに加え，ほかの利用者にとってもいくぶん理解しやすいモジュールでもある。

2.2 機能分解

機能分解とは，1つの大きいプログラムをいくつかの関数に分割していくことである。この手法は，プログラムが何をするものであるかという全体の表現から始め，各動作を複数の処理，つまり複数の抽象演算に分解していくことで行われる。各処理は，プログラム中の関数として実装される。この関数自体も，さらに小さい処理(これも関数として実装される)の集合へと精製する(細かく分解する)ことができる。最後に，さらに分解することなく実装できるほど精製されたとき，機能分解は完了する。

このデザイン法は，高レベルの関数が低レベルの関数に動作を委任するという，関数の階層構造を構成する。この手法は，プログラムの高レベルの表現で始まったデザインを，実装の低レベルの細部に向かって精製していくため，トップダウンデザイン法としても知られている。

例として，単語のリストをソートするためのプログラムを考えてみよう。まず，プログラム中で，リストと単語を表現する必要がある。単語のデータ構造は文字列(ヌルで終了し，ポインタによってアクセスされる文字の列)で表現し，リストは文字列へのポインタを要素とする配列で表現する。また，配列に格納できる文字列数の上限と，現在格納されている文字列の数を保存するための変数も別に用意する。

typedef 宣言を使って，文字列型“char *”に名前を与え，次にリストを作成し，空に初期設定する。

```
typedef char *String;

const int max = 100;
String list[max];

int size = 0;
```

これで関数をデザインする準備が整った。次の課題は，リストを受け取って，ソート済みリストを作り出すことである。最上位レベルとして，これを3つの演算に分解したものが考えられる。

```
オリジナルのリストを読み込み
リストをソートし
ソート済みのリストをプリントアウトする
```

したがって，main 関数は3つの関数を呼び出すようにすればよい。

```
int input( String *, int );
void sort( String *, int );
void output( String *, int );

main() {
    size = input( list, max );
    sort( list, size );
    output( list, size );
}
```

input 関数は，文字列を読み込み(ただしリストに格納できる上限まで)，リストに格納し，実際に読んだ文字列数を返す。sort 関数はリストの要素を順番に並べ換える。output 関数は，リスト中の各文字列を出力する。ここで，配列と要素数の情報が，関数への引数として与えられていることに注意しよう。これらの関数は，このプログラムで 1 回だけしか使われないので，これらの情報を引数としては渡さず，大域データを直接アクセスするようにすることもできただろう。しかし，これらの関数へ引数で情報を渡すようにすることで，これらの関数のメンテナンスが容易になり，またこれらをほかのプログラムからも再利用できるという長所が与えられる。

文字列の読み込み，文字列の表示，ソートに要する文字列の比較を除けば，input，sort，output の各関数の実装は非常に単純である。難しいところは低レベルの関数に委ねるものとし，あとで論じることにする。

```
int readString( String & );
void printString( String );
int lessthan( String, String );

int
input( String *a, int limit ) {
   for( int i = 0; i < limit ; i++ )
      if( !readString( a[i] ) )
         break;
   return i;
}

void
output( String *a, int size ) {
   for( int i = 0; i < size; i++ )
      printString( a[i] );
}

void
sort( String *a, int n ) {
   // 単純バブルソートアルゴリズムを使う
   int changed;
   do {
      changed = 0;
      for( int i = 0; i < n-1 ; i++ )
         if( lessthan( a[i+1], a[i] ) ) {
            String temp = a[i];
            a[i] = a[i+1];
            a[i+1] = temp;
```

```
            changed = 1;
        }
    } while( changed );
}
```

関数readStringは，文字列を読み出すことができれば1を返し，失敗すれば0を返す。関数lessthanは，アルファベット順で第1引数が第2引数より小さければ1を返し，違えば0を返す。

これで文字列を実際に処理する関数にたどりついた。ここで必要とされる関数は，あらかじめ標準ライブラリに用意されている。

```
#include <stdio.h>
#include <string.h>

int
readString( String &s ) {
    // 文字列を読み出して自分の領域に格納する。
    // 終了を意味するヌル文字のための領域も
    // 一緒に用意する必要があることに注意。

    const bufsize = 100;
    static char buffer[bufsize];

    if( scanf( "%s", buffer ) == EOF )
        return 0;
    s = new char[ strlen(buffer)+1 ];
    if( !s ) // newが失敗した場合
        return 0;
    strcpy( s, buffer );
    return 1;
}

void
printString( String s ) {
    printf(" %s", s );
}

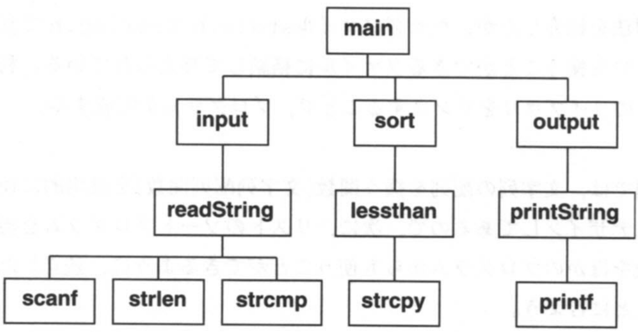
int
lessthan( String s1, String s2 ) {
    return strcmp( s1, s2 ) < 0;
}
```

関数readStringはリファレンス型の引数を持ち，新たに生成した文字列に引数の値を設定している。ここで“String &”の代わりにStringを引数の型として使用し

た場合，sへの代入は関数内部の局所変数の値を設定することを意味し，関数呼び出しの際に指定された実引数の値は変わらない。リファレンス型は，仮引数名という別名を通して，実引数の値を関数の内部で設定することを許す。

文字列の入出力は，標準ライブラリのヘッダファイルstdio.hで宣言されているscanf関数とprintf関数によって行われる。scanfはファイルの最後に達すると値EOFを返す。文字列に対してstrlenを実行すると，末尾のヌル文字を除いた文字の数を返す。strcpyはある文字列をほかの文字列に複写する。strcmpは，第1引数がアルファベット順で第2引数の前にあるか後にあるかに応じて負または正の値を返し，両者の文字がすべて同じであれば0を返す。これらの関数はstring.hで宣言されている。

このプログラムのトップダウンデザインは次の階層図のように表現できる。



2.3　ファイルの構成

C++のコンパイル単位(翻訳単位とも呼ばれる)はファイルである。C++の処理系は，ファイルの数がいくつであろうと，すべてをコンパイルし，リンクすることで，実行可能プログラムを作成することができる。宣言と定義で**外部リンケージ**を与えることによって，ファイル間相互で関数あるいはデータを参照することができる。通常，ファイル中の外部定義へのアクセスを与えるには独立したヘッダファイルを作成し，この**ヘッダファイル**には，外部宣言(extern)とともにファイル中の関数とオブジェクトを正しく使うために必要とされる型定義が書かれる。したがって，ヘッダファイル中のこれら

の宣言は，関係する外部関数または外部オブジェクトを使うあらゆるファイルにもインクルードされる。またstaticと宣言された関数と大域オブジェクトは**内部リンケージ**を持ち，宣言されたファイルの外部からは参照できない。また，constと宣言された大域オブジェクトは，明示的にexternと宣言しなければ，内部リンケージを持つ。

大きなプログラムをいくつかのファイルに分割し，リンケージをうまく使い分けることで，プログラムの開発，理解，メンテナンスの作業が容易になる。たとえば，関連する関数とデータ構造は，これを読みやすく理解しやすくするためにひとまとめにし，さらに外部に及ぼす作用を制限して局所的に修正することを許すことで，複数のプログラマが，相互に支障なく同時に同じプログラムに関する作業を行うことが可能になる。

いくつかのファイルに分割されたコードを取り出して，ほかのプログラムから再利用するのも容易である。これは，コードを再利用するための手段として代表的であり，ライブラリもこのようにして作成されている。すでに，標準入出力と文字列のライブラリの使用法を紹介したが，ヘッダファイルstdio.hとstring.hで宣言された関数は，だれでも使うことができるファイルに格納して与えられている。利用者のコードにこれらのライブラリをリンクすることで，プログラムが完成する。

先の例題では，文字列の配列を扱う関数(文字列配列関数)を汎用的に使うことができるようにデザインしてあるので，次に，リストのソートプログラムを再編成し，これらの関数をほかのプログラムからも使うことができるように，独立したファイルに格納することにしよう。

main関数をmain.ccという名前のファイルに格納する。本書では，C++プログラムのテキストファイルを示す約束として，ファイル名の最後の“.cc”に付けることにする。ファイル名の約束事は，C++のプログラミング環境によってまちまちである。ここで，ファイルスコープのデータをstaticと宣言したので，このファイルの外部からのデータアクセスは，関数インターフェイスに渡されるポインタを経由してのみ行われる。

```
#include "strarray.h"

const int max = 100;
static String list[max];

static int size = 0;
```

```
main() {
   size = input( list, max );
   sort( list, size );
   output( list, size );
}
```

ファイルmain.ccをコンパイルする際に，ファイルstrarray.hの内容がテキスト中の#include文の位置に挿入される。ヘッダファイルstrarray.hには，文字列配列関数を使うために必要とされる以下の宣言が書かれている。

```
typedef char *String;

int input( String *, int );
void sort( String *, int );
void output( String *, int );
```

このstrarray.hで宣言されている関数の定義はstrarray.ccに書かれる。また，関数を整理して見直す際に，printfStringはprintfがすでに用意している演算に何も付加していないため取り除いた。

```
#include <stdio.h>
#include <string.h>
#include "strarray.h"

static int
readString( String &s ) {
   // 前と同じ
}

static int
lessthan( String s1, String s2 ) {
   // 前と同じ
}

int
input( String *a, int limit ) {
   // 前と同じ
}

void
output( String *a, int size ) {
   for( int i = 0; i < size; i++ )
      printf( " %s", a[i] );
}
```

```
void
sort( String *a, int n ) {
    // 前と同じ
}
```

補助関数readStringとlessthanはstaticと宣言され，ほかのファイルからはアクセスできないように内部リンケージが与えられていることに注意しよう。これで，これらの関数はこのファイル中からだけ使われるように制限されるため，外部の利用者のことは無視して，このファイルの中のほかの関数の要求だけに従って変更したり，また削除することさえも可能である。

ここでstrarray.ccは，実装に必要とされるライブラリ関数が宣言された標準ヘッダファイルとともに，自分のインターフェイスのヘッダファイルstrarray.hをインクルードしている。これは，必要とされるString型の型定義を得ると同時に，宣言と実装の間に一貫性があるかどうかを検査するためである。

C++プログラムからC関数を呼び出すために，C関数はCリンケージを持つと宣言されている必要がある。

```
extern "C" int printf( const char *, ... );
```

stdioの関数はCの標準ライブラリの一部で，stdio.h中で“extern "C"”と宣言されている。中括弧を使ったリンケージ宣言を使うことで，Cリンケージの複数の関数をまとめて宣言することもできる。

```
extern "C" {
    int printf( const char *, ... );
    int scanf( const char *, ... );

    /* ほかのC関数 */

} /* extern "C"の終わり */
```

標準ヘッダファイルには，ライブラリ関数の正しいリンケージ仕様が書かれているので，通常，C++プログラマはCリンケージ宣言を扱う必要がないが，1つのプログラム中で各自のCとC++ファイルが混在している場合は，Cリンケージを与える必要がある。

2.4 構造化プログラミング

構造化プログラミングテクニックを使用すれば，関数の実装を理解しやすいものにすることができる。これには，規則正しい制御構造を利用して関数をブロックごとに分割し，スタイルを整備し，コメントを使ってコードを説明するといったことがある。

条件分岐と繰り返しの処理は，対応する制御文によって関数中の制御の流れを容易に追うことができるように実装される。これらの文は，制御構造に対して，規則的で容易に認識できる書式を与える。C++の制御文には，条件分岐(if，if-else，switch)と繰り返し(for，while，do-while)がある。このほかに，いくつかのジャンプ文も用意されている(繰り返しやswitch文からの抜けるためのbreak，次の繰り返し処理に移るためのcontinue，関数からの戻るためのreturn，悪名高いgoto)。goto文の存在は，関数中のあらゆるところに移動できるという汎用性ゆえに，関数中の逐次の演算を追いつつ理解することを難しくしてしまう。このため，goto文を使う際には，分岐または繰り返しを使って書き直すことができないかどうかを調べてみるべきである。

ジャンプ文の使用が問題視されるのは，制御の流れを不明瞭にしてしまうからだけではない。たとえば，goto文を使って，

```
    goto label;
    int i = 3;  // エラー!
  label:
    i++;
```

あるいは，制御構造で，

```
switch( x ) {
   int i = 0;  // 間違い!
case 1: {
   int j = 1;  // OK.
   break;
}
case 2:
   // etc.
default:
   break;
}
```

というように，ジャンプを使って，初期設定子を持つ宣言を飛び越えてスコープ中に入るように制御を移動させることは誤りである。初期設定子を持つ宣言の飛び越しが正

しいのは，宣言のスコープ全体を飛び越してしまう場合のみである。

すべての関数は，関数本体として最低１つのブロックを持ち，関数の仮引数宣言もこの本体ブロックの中にあると考えられる。また，ブロックは，ほかのブロックの中に入れ子にされてもよい。

ブロックは，制御構文の中の複数の文をひとまとめにするために使われることがほとんどであるが，使い道が限定されている名前を局所的するために使うこともできる。たとえば，関数sortでは，変数tempは使われているブロックの内部で宣言されている。

```
if( lessthan( a[i+1], a[i] ) ) {
    String temp = a[i];
    // etc.
}
```

この局所変数は，制御の流れが宣言位置を通過するときに初期設定され，制御が宣言を取り囲んだブロックを離れる際に，直ちに消滅する。もしtempが外側のブロックで宣言されていたとすると，tempの目的を宣言の文脈から明確に理解することはできず，宣言時には適当な初期設定値すら存在しないだろう。変数を局所ブロックスコープで宣言することのもう１つの長所は，ほかのスコープでも同じ簡略名を再び使うことができるということにある。

宣言は文でもあるから，これをブロック中の文を置ける場所であれば，どこにでも置くことができる。この柔軟性によって，最初に使われる位置で名前を宣言し初期設定することができる。たとえば，for文の１番目の節は式文であるから，繰り返し処理のインデックスをこの位置で宣言することもできる。

```
for( int i = 0; i < limit ; i++ )  {
    // etc.
}
return i;
```

名前のスコープは，宣言した位置から宣言を取り囲んだブロックの最後までである。したがって，この例のiは，ループ終了後でもまだ有効であることに注意されたい。スコープは，for文が格納されているブロックの最後で終了する[1)]。

---

1) 訳注：新しい標準規格案では，for文の第１節で宣言された変数のスコープがfor文の最後で終了するように変更された。付録B.3参照。

コードのスタイルを整理することを癖にし，関数を読みやすく書くことは重要である。関数の制御構造を明らかにするためには，文とブロックの入れ子関係を示すためのインデント(字下げ)が最も重要である。また，よく検索される部分は目立つようにすべきである。たとえば，宣言，関数ブロックの始まりと終わり，およびラベルは，はっきりと見えるようにするとよい。コード部を強調するためにインデントレベルを変えたり，前後を空白にしたり，行の先頭やブロックの始まりの配置を工夫することもよいだろう。また，コメントによって，コード部分を指示することもできる。

C++には，コメントを指示するために2つの書式がある。ブロック書式のコメントは“/*”で始まって“*/”で終わる。行書式のコメントは“//”で始まって行の最後まで続く。コメントは，関数あるいはコードの入り組んだ部分の概略を説明するために使うことができるし，巧妙かつ秘密の意味を持つ文を説明することもできる。たとえば，実装時に行った暗黙の仮定とか，各種の演算間に存在する内密の依存関係も示されるべきである。

```
/*
    大域変数 flag1 は，後の func2 呼び出しを左右する
    文脈設定に使われるので，func1 が呼び出す前に
    設定される必要がある。
*/

// ASCII 文字集合を仮定
```

コメントは，不規則性を明らかにするためにも使うべきである。たとえば，

```
// これはつぎあてである
```

といったコメントは変則のバグ修正情報を与えているが，実際にはほとんど無意味である。コメントにはできるだけ情報を盛り込んだほうがよい。

```
// 4捨5入誤差の特別補正
```

関数に注釈を付けるという意味では，関数名と変数名に自然と使い道がわかるものを選択することが最も重要である。特に，関数名は，先の例の`input`，`sort`，`output`のように，各関数が実現している抽象演算の意味を示すようにすべきである。プログラム中で，意味のある，覚えやすい名前を使うようにすれば必要に応じてコメントを追

加するだけよいが，名前が無意味であればいくらコメントを追加したところで，コードは理解しにくいままだろう。

これらの議論は，あくまでも自分のコードをほかのひとに理解してもらうことを望んでいるという仮定の上に成り立っている。将来的に再使用されることを考慮に入れずに問題を解決するプログラムを書くこともあるし，ほかのひとが変更したり再利用したりできないように，プログラムを意図的に不明瞭にしてしまうこともある。名前をわざと取り換え，コメントを取り除き，スタイルを崩すといった手段を駆使すれば，競争相手がコードを盗もうとしてもたいへん高いものについてしまうだろう。しかし，公然と敵意を示したいときを除けば，わかりにくいコードを書いたりコメントを取り除いたりするという行為は間違っている。ほとんどのプログラムは，メンテナンスが簡単かどうかにかかわらず，最後にはメンテナンスが必要である。わかりにくいプログラムを損傷させずに改変しつつ更新していくのは難しく，メンテナンスの費用も高騰してしまうだろうし，最終的にはプログラムを処分するという結果が待っているだろう。プログラマは，自らの仕事を無駄にしないためにプログラムを理解しやすいものにすべきである。

2.5　関数の多重定義とインライン関数

これまでに，関数が抽象演算の実装にどのように使われるか，関数名が演算の情報をどのように提供するかを論じてきた。これらは，関数名の**多重定義**と**インライン関数**という機能によってさらに強力にサポートされる。

引数の型と数によって複数の実装を区別することさえできれば，多重定義機能によって，複数の関数の実装に同じ名前を使うことができる。複数の関数を区別するために無理に名前をわざとらしく考え出さなくても，このようにして同じ抽象演算を複数のオペランド型に対して実装することができる。

たとえば，2種類のリスト型に対して動作するように，sortを多重定義することもできる。

```
typedef char *String;

void sort( String *, int );
void sort( int *, int );
```

1番目の実装は文字列の配列をソートし，2番目の実装は整数の配列をソートする。次の関数呼び出し

```
sort( list, size );
```

は，list の型が“String *”であれば1番目の実装を，また“int *”であれば2番目の実装を呼び出す。

抽象演算を実装するためにたった数行の文を関数をまとめ，これを呼び出すことは，効率が悪いという理由でプログラマから嫌われている。インライン関数機能は，こういった小さい関数をサポートするために与えられている。関数宣言にinline指定子を付けると，指定された関数の呼び出しを関数本体でインライン展開すれば，効率を改善できるということがコンパイラに指示される。インライン展開では，関数呼び出しの意味が維持される。あらゆる関数(main関数を除く)をinlineと宣言してもよいが，実際にインライン展開されるかどうかは，コンパイラの実装と呼び出し文脈に依存する。インライン関数を呼び出すことができるのは，staticと宣言された関数と同じで，対象の関数が定義されているファイルの中だけである。この理由は，インライン展開を行うためには，関数を呼び出している場所から関数定義が見えている必要があるためである。したがって，複数のファイルで使われるinline関数は，ヘッダファイルで定義するべきである。

通常の関数であれば，関数の呼び出し側は，関数のインターフェイスに変更があったときだけ再コンパイルすればよいのであるが，インライン関数を使っているコードは，関数の内容に変更が加えられただけであっても，再コンパイルされる必要がある。これは，インライン展開による性能改善の結果，コンパイル済みコードのモジュール性の損失がもたらされることを意味する。

先の文字列比較関数lessthanの本体は，たった1つの文だけであるから，必然的にインライン関数の候補である。

```
inline int
lessthan( String s1, String s2 ) {
    return strcmp( s1, s2 ) < 0 ;
}
```

次に示すリストのソート例題の拡張版では，関数の多重定義機能とインライン関数lessthanを使用し，最初に文字列のリストをソートし，次に数値のリストをソートしている。ここでもリストを配列として実装するが，今度は数値型要素の配列と文字列型要素の配列を１つづつ用意する。

```
#include "arrays.h"

const int max = 100;
static String strlist[max];
static int intlist[max];

static int size = 0;

main() {
    // 文字列のリストを読み込み，ソートし，書き出す
    size = input( strlist, max );
    sort( strlist, size );
    output( strlist, size );

    // 数値のリストを読み込み，ソートし，書き出す
    size = input( intlist, max );
    sort( intlist, size );
    output( intlist, size );
}
```

ヘッダファイルarrays.hでは，配列のための多重定義関数の宣言が行われる。

```
typedef char *String;

int input( String *, const int );
int input( int *, const int );

void output( String *, const int );
void output( int *, const int );

void sort( String *, const int );
void sort( int *, const int );
```

関数本体はarrays.ccの中に書かれる。２つの多重定義関数の実装をみると，両者は非常に似ていることがわかるだろう。これは関数を再利用するための手段である，「コードの流用と編集」のよい例である。この例では，この手段で作られたことを明らかにするため，２つの関数を作成するために汎用の関数テキストのテンプレートを使い，型の違いに対処するためのただの文字列置換のみを行った。

```
#include <stdio.h>
#include <string.h>
#include "arrays.h"

static int
readString( String &s ) {
   // 前と同じ
}

inline int
lessthan( String s1, String s2 ) {
   return strcmp( s1, s2 ) < 0 ;
}

int
input( String *a, const int limit ) {
   for( int i = 0; i < limit ; i++ )
      if( !readString( a[i] ) )
         break;
   return i;
}

int
input( int *a, const int limit ) {
   for( int i = 0; i < limit ; i++ )
      if( scanf( "%d", a+i ) == EOF )
         break;
   return i;
}

void
output( String *a, const int size ) {
   printf( "\nlist:" );
   for( int i = 0; i < size; i++ )
      printf( " %s", a[i] );
   printf("\n");
}

void
output( int *a, const int size ) {
   printf( "\nlist:" );
   for( int i = 0; i < size; i++ )
      printf( " %d", a[i] );
   printf( "\n" );
}
```

```
void
sort( String *a, const int n ) {
   int changed;
   do {
      changed = 0;
      for( int i = 0; i < n-1 ; i++ )
         if( lessthan( a[i+1], a[i] ) ) {
            String temp = a[i];
            a[i] = a[i+1];
            a[i+1] = temp;
            changed = 1;
         }
   } while( changed );
}

void
sort( int *a, const int n ) {
   int changed;
   do {
      changed = 0;
      for( int i = 0; i < n-1 ; i++ )
         if( a[i+1] < a[i] ) {
            int temp = a[i];
            a[i] = a[i+1];
            a[i+1] = temp;
            changed = 1;
         }
   } while( changed );
}
```

多重定義関数を使用するにあたっては注意が必要である。多重定義機能は，プログラム中の複数の関数の実装を概念的に統一する代わりに，関数名に曖昧さを持ち込んでしまう。先の多重定義関数sort の例で，

```
sort( list, size );
```

という呼び出しでは，呼び出される関数を決定することはあまり難しいことではない。この呼び出しでは，引数型が自動的に変換されることがないので，listの宣言で与える型が，2 つの関数のどちらを呼び出すべきかを直接示している。多重定義関数が複数あり，引数間の変換がいくつか用意されているとすると，呼び出される関数の決定は難しい。また，関数名の曖昧さにより，プログラムが実際に何をするのかがわかりにくいこともあるということに留意すべきである。

2.6 テンプレート関数

先に示した多重定義関数sortは，同じアルゴリズムのほとんど同一の実装である。各関数のテキスト上での違いは，宣言と演算で引数の型を明らかにする必要がある部分だけである。これらの関数が同一の演算を用意した類似の型に対し多重定義されているのであれば，これらの違いは型宣言の部分だけである。この場合，C++の関数テンプレートを使えば，複数の多重定義関数を定義する代わりに，類似の型に対して関数定義を1つだけ与えることができる。

以下に，arrays.hで，テンプレートを使って多重定義関数を宣言した例を示す。

```
template<class T> int input( T *a, int const limit );

template<class T> void output( T *a, const int size );

template<class T> void sort( T *a, const int n );
```

“<class T>”はテンプレートパラメタ並びであり，ここでは，予約語classはTの型名であり，このTが，テンプレートのインスタンス(テンプレート関数と呼ばれる)を作成する際に実際の型のための予約席の役目を果たすことを示している。つまり，これらのテンプレート宣言は，input，output，sortの多重定義関数のインスタンスの集まりを宣言する。これには当然intとStringに対するインスタンスも入っている。したがって，これらの関数のどれかが呼び出された場合，多重定義解決の際に，実引数にマッチする型の関数を作成するためにテンプレート定義の展開が行われる場合もある。

次のテンプレート機能を使って，これらの関数のint版を基本にし，あらゆる数値型に使うことのできるテンプレート定義を定義してみよう。

```
template<class T> int
input( T *a, int const limit ) {
  double temp;
  for( int i = 0; i < limit ; i++ )
   if( scanf( "%lf", &temp ) == EOF )
      break;
   else {
      a[i] = (T)temp;
      if (a[i] != temp) {
         printf("error: input %f truncated\n",
            temp);
         a[i] = 0;
```

```
      }
    }
  return i;
}

template<class T> void
output( T *a, const int size ) {
   printf( "\nlist:" );
   for( int i = 0; i < size; i++ )
      printf( " %f", (double) a[i] );
   printf( "\n" );
}

template<class T> void
sort( T *a, const int n ) {
   int changed;
   do {
      changed = 0;
      for( int i = 0; i < n-1 ; i++ )
         if( a[i+1] < a[i] ) {
            T temp = a[i];
            a[i] = a[i+1];
            a[i+1] = temp;
            changed = 1;
         }
   } while( changed );
}
```

これらの関数定義は，Tと取り換えることができる数値型に対して使うことができる。ただし，scanfとprintfの書式指定は型ごとに違うので，ここでは，doubleを使って入出力を行い，doubleと配列要素の型の間に変換を課している。double型は，ほかのあらゆる数値型の値の範囲を表現できる。また，この代案として，すべての数値型に対して使うことのできる入出力関数を多重定義関数を使って用意することもできる。

ここで，上で定義した関数テンプレートを使って，いくつかの数値型の配列をソートしてみよう。

```
#include "arrays.h"

const int max = 100;
static double dlist[max];
static short slist[max];
static String strlist[max];
```

```
static int size = 0;

main() {
    // double型の整数のリストの読み込み，ソート，書き出し
    size = input( dlist, max );
    sort( dlist, size );
    output( dlist, size );

    // short型の整数のリストの読み込み，ソート，書き出し
    size = input( slist, max );
    sort( slist, size );
    output( slist, size );

    // 文字列のリストの読み込み，ソート，書き出し
    size = input( strlist, max );
    sort( strlist, size );
    output( strlist, size );

    return 0;
}
```

dlistを引数とする関数呼び出しは関数テンプレートにマッチし，テンプレートが展開され，以下のテンプレート関数が作成される。

```
int input( double *a, int const limit );
void output( double *a, const int size );
void sort( double *a, const int n );
```

これと同じように，slistを引数とする関数呼び出しでも，以下の関数が生成される。

```
int input( short *a, int const limit );
void output( short *a, const int size );
void sort( short *a, const int n );
```

“char *”とdouble間の型変換は存在しないので，呼び出し“input(strlist, max)”は関数テンプレート定義にはマッチせず，strlistを引数とした呼び出しは，String版の多重定義関数と解決される。

テンプレート関数の欠点は，実引数を仮引数型にマッチさせるための型変換が行われる前に，多重定義関数の呼び出しを解決するための暗黙のテンプレート展開が行われてしまうことにある。つまり，型変換後の引数で既存の関数を呼び出してほしい場合であっても，関数テンプレートのインスタンスが生成されてしまうことがある。た

とえば，以下のテンプレート関数宣言を仮定すれば，

```
template<class T> void f( T );
```

次の2つの呼び出し

```
int i;
short s;

f( i );
f( s );
```

は2つの関数インスタンス“f(int)”と“f(short)”を生成してしまう。ここで，関数テンプレートの存在を無視すれば，以下の宣言の関数1つで，2つの関数呼び出しを処理することができる。

```
void f( int );
```

この場合，引数sは仮引数のint型にマッチするように自動的に変換される。

テンプレート関数は，総称クラス型と一緒に使うと特に便利である。これは第4章で解説する。

2.7　引数と戻り値

関数の型は仮引数の型と戻り値の型から構成される。関数呼び出しでは，呼び出すべき関数はまず名前によって識別され，次に呼び出し時の引数の型が，宣言されている仮引数型とマッチするかどうか，あるいは利用できる型変換を適用することでマッチさせることができるかどうかが調べられる。最後に，いくつかの多重定義関数の中から，引数の型に仮引数の型がマッチするものが選択される。

引数の型と数が可変である関数の型は，仮引数並びの最後の“...”(省略符号)によって示される。

```
void error( const char *, ... );
```

この宣言は，関数errorが値を返さず，第1引数には文字列をとり，残りの引数として個数と型が可変の引数をとることができることを宣言している。ここでの“const”は，第1引数のポインタで渡される文字列の変更を禁止しているので，この関数の呼

び出しでは文字列定数は安全に扱われる。たぶん，error の第１引数は，後続の引数を解釈するための情報を与えるものだろう。

引数は，関数の呼び出し時に自動的に仮引数の型に型変換される。仮引数がクラス型である場合は，第１章のクラスcomplex の例で示したように，引数をクラスのコンストラクタの仮引数型にマッチさせるために，言語定義の型変換があらかじめ適用されることもある。クラス型の引数を非クラス型の仮引数へと型変換する場合は，利用者定義の型変換のあとで，言語定義の型変換が追加して適用されることもある。省略符号で指定された仮引数の場合，int より小さい整数型はすべてint 型あるいはunsigned 型に変換され，浮動小数点型はすべてdouble 型に変換される。このほかの引数は型変換されない。

関数の実行時には，関数定義の仮引数は関数呼び出しの実引数によって初期設定される。また，関数からの戻り値は，結果を初期設定するために使われる。このように，引数と戻り値は代入されるのでなく初期設定されるのだという事実は，初期設定と代入の意味に特に違いがあるコンストラクタを持つクラス型だけでなく，const 型とリファレンス型にとっても，特に重要である。

関数の仮引数は，関数ブロック内部の局所変数である。仮引数は，いったん関数呼び出しによって初期設定されると，ほかのあらゆる変数と同じように使うことができ，代入や値の変更も可能である。限定子const は，const と宣言されたオブジェクトの値は，初期設定された後では，変更することができないことを意味する。したがって，const と宣言された仮引数の値は，引数として渡された最初の値から変わることはない。つまり，この仮引数宣言は，値を固定したしたいものを誤って変更しないようにするためのものである。たとえば，リストソートの例題では，ソートすべきリストのサイズは固定である。したがって，これを示すために仮引数をconst 宣言することができる。

```
sort( String *, const int );
```

リファレンス型仮引数を除けば，仮引数と実引数のオブジェクトは別物である。仮引数は，いったん引数の値で初期設定されると，あとは関数の外部に何も影響を与えずに，仮引数を局所的に使ったり代入したりすることができる。第１章で述べたように，リファレンス型仮引数は実引数から独立したオブジェクトではない。リファレンス

型仮引数の初期設定では，初期設定子であるオブジェクトを参照するための別名としてリファレンス名が作成される。リファレンス型仮引数による「リファレンス呼び出し」を行うと，関数内部での仮引数への代入は実引数の値の変化させることを意味する。たとえば，readStringでは，引数を通して文字列を返すために，リファレンス型の仮引数が使われていた。

```
int readString( String & );
```

C++には，柔軟に関数を使うための機能として**デフォルト引数**がある。これは，関数宣言において引数のデフォルト値を与えておけば，関数呼び出しの引数の数が関数宣言での引数の数に不足したときには，不足分を自動的に補充させることができるというものである。たとえば，以下に示した関数は 3 個の引数を持つように宣言されているが，このうち 2 個はデフォルト値を持つ。

```
int
error( const char *msg, int level = 0, int kill = 0 );
```

この関数は，引数の数が 1 個，2 個，または 3 個のいずれでも呼び出すことができる。

```
error( "you goofed" );
   // 実際の呼び出しは error( "you goofed", 0, 0 );
error( "you screwed up", 1 );
   // 実際の呼び出しは error( "you screwed up", 1, 0 );
error( "you blew it!", 3, 1 );
   // 引数のデフォルト値は使われない
```

引数のデフォルト値は，メンテナンスの期間中にプログラムに加える変更の影響を最小限にとどめるために特に便利である。つまり，既存の関数を引数を増やしたいときにも，既存の関数呼び出しのソースコードを変更する必要はなく，宣言に新しい仮引数のためのデフォルト値を与えることで，古い関数呼び出しが以前のまま存在することが許される。したがって，関数の動作を容易に拡張することができるわけである。

例として，先の文字列配列のソート関数を，アルファベット順と逆順のどちらででもリストをソートできるように変更してみよう。最初に，ヘッダファイルの関数宣言に新しい引数の型とデフォルト値を付け加える。

```
void sort( String *, int, int descending = 0 );
```

関数宣言中の引数名descendingはただの装飾であり，読者にどうしてこの引数が存在するかということを知る手がかりを与えるコメントとしての意味しか持っていない。このヘッダファイルを使用しているファイルを再コンパイルすると，文字列配列のソート関数の呼び出しのうち，引数を2個だけ持つ関数呼び出しすべてには，第3引数として0が自動的に加えられる。

次に，新たに追加した第3引数を利用するようにソート関数を変更する。最初に，2つの文字の比較を行う関数としてgreaterthanを追加する。さらに，lessthanとgreaterthanのどちらでも呼び出せるようにするため,「関数へのポインタ」変数を使うようにsort関数を変更する。この結果，関数呼び出しはポインタを通して行われるので，ここではinline指定子は無意味である。

```
inline int
lessthan( String s1, String s2 ) {
   return strcmp( s1, s2 ) < 0 ;
}

inline int
greaterthan( String s1, String s2 ) {
   return strcmp( s1, s2 ) > 0 ;
}

void
sort( String *a, int n, int descending ) {

   int changed;
   typedef int (* Fptype)( String, String );

   // 比較関数へのポインタを宣言し，初期設定する
   Fptype compare = descending ?
            &greaterthan : &lessthan;

   do {
      changed = 0;
      for( int i = 0; i < n-1 ; i++ )
         if( compare( a[i+1], a[i] ) ) {
            String temp = a[i];
            a[i] = a[i+1];
            a[i+1] = temp;
            changed = 1;
         }
   } while( changed );
}
```

比較関数を選択するために使う“? :”演算子は条件式である。第1オペランドdescendingが0以外の値であれば，第2オペランド&greaterthanを評価したものが式の結果であり，descendingが0であれば，第3オペランド&lessthanを評価したものが式の結果である。

文字列配列のソート関数は，変更されたヘッダファイルをインクルードしているファイルを再コンパイルする必要があることを除けば，この関数の既存の呼び出しに何ら影響を与えることなく拡張された。関数sortの新しい呼び出しでは，第3引数を設定することで新しい機能を利用することができる。

```
#include "arrays.h"

const int max = 100;
static String strlist[max];

static int size = 0;
const int descending = 1;

main() {
    // 前と同じようにソートする
    size = input( strlist, max );
    sort( strlist, size );
    output( strlist, size );

    // 今度は降順でソートする
    size = input( strlist, max );
    sort( strlist, size, descending );
    output( strlist, size );
}
```

演　習

演習 2-1　文字列ソートのプログラムを修正して，任意の個数の文字列を扱うことができるようにせよ。最初の解法から，関数の数や依存関係を変更してはならない。これは，この問題に対する別の解法だろうか，あるいは単に同じ解法のもう1つのバージョンだろうか。

演習 2-2　文字列のソート問題を解くための，次に示す概念に対応したもう1つの解を見つけよ。

```
繰り返し
    文字列を１つ読み出して正しい位置に挿入する
文字列をプリントする
```

どちらの解法が効率がよいだろうか。対話型のアプリケーションには，どちらの解法が実装しやすいだろうか。どちらの解法のほうが優れているだろうか。

演習 2-3 ある種の関数は，通常，プログラム中の別の関数から便利に使われることが多い。たとえば，error 関数は，エラーメッセージを出力すべきときに，いつでも利用することができる。またlookup 関数は，格納した情報へのアクセスを必要とする，あらゆる関数から使うことができる。階層的にデザインを行う機能分解法は，この種の汎用のユーティリティ関数の識別をどのように妨げるだろうか。

演習 2-4 トークンとは，空白，タブ，または改行といった区切り文字で囲まれた，テキスト中の単語あるいは意味のある文字の集まりのことである。引数として与えられた文字列中の次のトークンのアドレスを返すと同時に，引数の値をこのトークンの終わりの次の１文字を指すように変更する関数を書け。この関数には，トークンがどの文字で区切られるかを指定するオプションとして，第２引数を持たせよ。引数文字列の長さを返す関数strlen を書くために，この関数を使用してみよ。

演習 2-5 ソート済みのint の配列に対して，int のキーを使った２分探索を行う関数

```
int bsearch( int *array, int key, int num );
```

を書け。さらに，a) double のキーを持つdouble の配列，b) int のキーを持つ構造体の配列，c) 文字列のキーを持つ構造体の配列，d) int のキーを持つ構造体へのポインタの配列，e) int のキーを持つ(全部をメモリに読み出すには大きすぎる)ディスク上の構造体の配列に対して動作するように，このbsearch を書き直せ。

演習 2-6 次のプログラムを考えよう。

```
void f( int ) {}
void f( long ) {}

main() {
    f( 40000 );
}
```

各自のC++処理系は，多重定義関数f のどちらの実装を呼び出すだろうか。また，このC++プログラムをほかのコンパイラでコンパイルした場合，違った動作することがある。これも正しい動作であるが，この理由を説明せよ。

演習 2-7　以下のswap 関数テンプレートの賛否両論を議論せよ。

```
template <class T>
void
swap( T &a, T &b ) {
    a ^= b;
    b ^= a;
    a ^= b;
}
```

3

クラス



関数とデータ構造は相互に依存している。関数は特定のデータ構造を処理するようにデザインされ，対応するデータ構造は，これらの関数から使用されても常に正しい状態を保つようになっていなければならない。第2章の文字列ソートの例では，関数は，文字列を表現するヌルで終了する文字配列というデータ構造に依存している。ある関数が終了のヌルを誤って上書きしてしまえば，ほかの関数はこの壊れたデータ構造に対しては正しく動作しなくなるだろう。また，文字列の表現方法が，終端文字を使用する構造から長さ情報を明示的に持つものに変更されたとすると，この新しいデータ構造に対して動作する関数群を新しく用意し直す必要がある。プログラムが大きくなるにしたがって，こうした多数の関数と入り組んだデータ構造の依存関係を管理することは難しくなる。

C++のクラス機能は，プログラムを構築したり，データ構造と関数の依存関係を管理したりするための多くの手段をサポートする。このクラスを使ったプログラミングテクニックが，本書の以降のほとんどの部分での話題である。この章では，これら後の議論の土台として，C++のクラスの特徴を紹介する。

3.1　クラス型

クラスは利用者定義の集合データ型である。クラスは，型を表現するデータメンバと，型に演算を実装するメンバ関数の両者から構成される。クラスは，内部を部分的にprivate(非公開)部として隠したり，逆にpublic(公開)部として利用可能であることをはっきりと示したりすることができる。private部のメンバには，そのクラスのメンバ関数と，そのクラスから“friend(フレンド)”であると宣言された関数のみがアクセスできる。クラスは，この情報隠蔽の機構を使って，クラスによって特定された関数だけがそれを扱えるように，データ構造を覆い隠してしまうことができる。

文字列ソートの例を，文字列を表現するクラスStringを使って実装し直してみよう。クラスのタグ名“String”は，このクラス型の名前であり，これはtypedefによって定義された型の名前と同様に使用することができる。

```
#include <stdio.h>
#include <string.h>

class String {
    char *str;
  public:
    String() { str = new char[1]; *str = '\0'; }
    String( const char * );
    void print() { printf( "%s", str ); }
    friend int operator <( const String &s1,
                    const String &s2 )
        { return strcmp( s1.str, s2.str ) < 0; }
};
```

文字列を表現するための基本のデータ構造は同じ(依然としてヌルで終了する文字の配列へのポインタ)であるが，今度はデータ構造自体はStringのpravate部のメンバstrとして隠され，コンストラクタString，メンバ関数print，フレンド関数“operator <”だけが，この文字列表現をアクセスできる。これらのメンバ関数はすべてクラスのpublic部で宣言されており，どこからでも使うことができる。

最初の2つのメンバ関数はクラスと同じ名前を持っている。このことは，これらのメンバ関数がこのクラスのコンストラクタであることを示している。クラスStringのオブジェクトが生成されるときには，初期設定のために常にコンストラクタが使われる。最初のコンストラクタStringは引数を取らず，strが空の文字列を指すように初期設定する。演算子newは，文字列を格納する領域を割り当てるために使われている。この関数定義は短いので，本体をクラス定義の内部で定義している。これはinline指定のメンバ関数とフレンド関数を作成するための簡略法である。

2番目のコンストラクタは，クラス内部では宣言が行われているだけであり，本体の定義をどこか別のところに用意する必要がある。

```
String::String( const char *s ) {
    if (s) {
        str = new char[ strlen(s)+1 ];
        strcpy( str, s );
    } else {
        // ヌル文字列の作成
        str = new char[1];
        *str = '\0';
    }
}
```

メンバ関数の定義をクラス外部で行う場合は，目的の関数がクラスのスコープの中にあることを示すために，関数名を修飾する必要がある。ここでは，スコープ限定子String::によって，クラスStringのメンバ関数の定義であることを示している。

メンバ関数は，常にそのクラスのスコープの中にある。したがって，メンバ関数は，メンバアクセス演算子なしでクラスオブジェクトのメンバを参照できる。ここで，コンストラクタStringとメンバ関数printでは，スコープを限定せずにstrを参照していることに注意しよう。これに対して，フレンド関数の場合は，

```
friend int operator < ( const String &s1, const String &s2 )
   { return strcmp( s1.str, s2.str ) < 0; }
```

というように，String引数のメンバstrにアクセスするために，メンバアクセス演算子“.”を使う必要がある。フレンドであれば，privateメンバstrをエラーなくアクセスすることができるが，関数のスコープはフレンドでないときと同じである。

演算子関数は，言語定義の演算子をクラス型のオペランドにも適用できるように多重定義するためのものである。ここでは，2つのStringリファレンス型の引数を比較する“<”(「小なり」)演算子が定義されている。これで，この中置形式の演算子“<”を，2つのString型のオペランドを比較するためにも使用できるようになる。

文字列ソートの例題を完成させながら，型Stringがどのように使われるかを見ていこう。

```
void
input( String *a, int limit, int &i ) {
   char buffer[100];
   for( i = 0; i < limit; i++ )
      if( scanf( "%s", buffer ) == EOF )
         break;
      else
         a[i] = String( buffer );
}
```

関数inputは，文字列を読み込み，リストへ格納する。

```
a[i] = String( buffer );
```

という文は，入力バッファの文字からクラスStringのオブジェクトを作成するためにコンストラクタを実行するという処理を通じて型変換を行う。このStringオブジェクトは，リスト配列の要素に代入される。

リストを出力する手続きでは，リスト中の各要素に対して，クラスStringの出力関数printがメンバアクセス演算子“.”を使って呼び出される。

```
void
output( String *a, int size ) {
   printf( "\nlist: " );
   for( int i = 0; i < size; i++ )
      a[i].print();
   printf( "\n" );
}
```

リストをソートする関数では，リスト中の要素を比較するために“operator <”が中置形式を使って呼び出されているので，リスト要素の型を除けば，この手続きは第2章の数値リストに関するソートと同じもののように見える。

```
void
sort( String *a, int n ) {
   int changed;
   do {
      changed = 0;
      for( int i = 0; i < n-1; i++ )
         if( a[i+1] < a[i] ) {
            String temp = a[i];
            a[i] = a[i+1];
            a[i+1] = temp;
            changed = 1;
         }
   } while( changed );
}
```

最後にmain関数が，リストを読み出し，ソートし，出力する。

```
const int max = 10;
static String list[max];

static int size = 0;
```

```
main() {
    input( list, max, size );
    sort( list, size );
    output( list, size );
}
```

これは，第2章で，型Stringを“char *”を使って実装したものとまったく同じに見えるが，実は違いがある。ここでは，配列listの各要素を初期設定するために，引数を取らないコンストラクタが呼び出され，リスト配列の各要素は，ただの0ではなく，Stringを表現する正しい値で初期設定されている。このことは，このプログラムにとって重要ではないが，オブジェクトの中に覆い隠されたデータ構造は自動的に正しい状態で始まるので，コンストラクタによるクラスオブジェクトの自動初期設定は，プログラムの動作をより正確にするということを意味する。

3.2　データメンバ

前節のStringのようなクラス型の定義は，目的のクラス型のオブジェクトを生成するための雛型として機能する。クラス型のオブジェクトの生成時には，まずオブジェクトのための記憶領域が割り当てられ，さらに，各データメンバがオブジェクトの構成要素として作成されて初期設定される。クラス型のオブジェクトは，C++のほかのデータオブジェクトと同じように，いくつかの違うスコープの中に作成でき，存続期間もまちまちである。ファイルスコープの静的オブジェクトは，プログラムの実行が開始される前に作成され，プログラムが終了するまで存続する。局所スコープの静的オブジェクトは，最初に使われるまでに初期設定され，プログラムの終わりまで存続する。仮引数とこのほかの局所オブジェクトは，プログラムの実行がオブジェクトの宣言位置に達したときに作成され，宣言されているブロックが終了するまで存続する。演算子newを使って動的に作成されたオブジェクトは，演算子deleteを使って明示的に削除されるまで存続する。また，式評価の中間結果あるいは関数の戻り値を保持するために，一時オブジェクトが作成されることもある。

クラスオブジェクトを構成するメンバにアクセスするには，メンバアクセス演算子“.”あるいは“->”を使う。演算子“.”は，クラスオブジェクトとともに使われる。

```
String s;
s.str;
```

演算子“->”は，クラスオブジェクトへのポインタとともに使われる。

```
String *sp = new String;
sp->str;
```

演算子newは，引数で指定された型のオブジェクトを生成し，この新しいオブジェクトへのポインタを返す。

クラスの各データメンバの型は，言語定義の型，すでに定義されているクラス型，クラス型へのポインタ，あるいはクラス型へのリファレンスのどれでもよい。クラス型へのポインタあるいはリファレンスを使う場合は，目的のクラスが前もって定義されている必要はなく，クラス名だけが宣言されていればよい。

```
class Node; // 名前だけを宣言する
Node *np;   // ポインタを宣言するためにクラス名を使う
```

クラス型へのポインタあるいはリファレンスをクラスのメンバとすることで，クラスの構造を再帰的に構築することが可能である。たとえば，二分木の節は，左右の子孫へのポインタをメンバとして持つだろう。

```
class Node {
    Node *left, *right;
    // etc.
};
```

ここでは，これらはポインタあるいはリファレンスでなければならないことに注意しよう。Nodeは直接Node自体をメンバとして持つことはできない。

クラスオブジェクトの各データメンバには，各型の表現に必要とされる大きさの領域が割り当てられる。このとき，アラインメント調整を必要とする機種では，クラスオブジェクトのメンバとメンバの間に詰め物が存在することがあるだろうし，以降の章で解説する機能を実現するために，コンパイラが隠れた(暗黙の)データメンバを追加することもある。

クラスオブジェクト内部の領域を節約する手段の1つとして，同じ領域を2つ以上のメンバに共用させることがある。これを実現するクラスでは，オブジェクト中のデータメンバは，連続して配置されるのではなく，すべてのデータメンバはオブジェクトの最初の位置から重複して配置される。この種のクラスは**共用体**と呼ばれる。共用体型

の宣言では，classの代わりにunion指定子が使われる。この共用体型のメンバはすべて，デフォルトでpublicである。

```
union Un_type {
   int i;
   double d;
   char *p;
};
```

共用体型のオブジェクトの生成の際に，すべてのデータメンバがオブジェクト中で同じ位置を占めるように，オブジェクトの大きさとアラインメントが調整される。共用体のメンバは，通常のメンバアクセス演算子を使って参照することができる。共用体メンバをアクセスすることで得られるものは，共有されている領域にあるデータを，メンバの型に従って解釈した結果である。

```
Un_type u;

u.i = 1;          // intの値をuに格納した
u.d = 3.1415;     // doubleの値でintの値を上書きする

char c = *u.p; // 危険！ ポインタの値はたぶん正しくない
```

共用体メンバに，ある型の値を格納し，この値をほかの型としてアクセスすることには危険がある。あるメンバにとっては正しい値が，ほかのメンバにとっても正しいとの保証はない。共用体を正しく使用するためには，どのメンバを使うべきかを示す情報が常に入手できるようにしておく必要がある。このため，共用体をクラスのメンバとするのであれば，この共用体のメンバをどのように解釈すればよいかという情報を与えるメンバを別に用意すべきである。

```
enum TypeCode{ is_int, is_double, is_charstar };

class Node {
  public:
   Node *left, *right;
   TypeCode code;
   Un_type info;
};
Node anode, *np = &anode;
```

この例では，Nodeオブジェクト中のcodeが，共用体メンバを解釈する情報を与えている。

```
int i = 0; double d = 0; char *p = 0;

switch( np->code ) {
   case is_charstar:
      p = np->info.p;
      break;
   case is_double:
      d = np->info.d;
      break;
   case is_int:
      i = np->info.i;
      break;
}
```

Nodeは一度に1つの型の情報しか持たないため，複数の型のメンバを重ねた共用体を使うことで，Nodeオブジェクト中の領域を節約することができる。

クラスの中に名前なしの共用体を入れ子にすることで，クラスメンバを共用体メンバと同様の手法で重複させることもできる。このとき，クラスメンバへのアクセスにおける共用体メンバアクセス記法の付加は不要になる。名前なしの共用体は，クラスタグを持たず，メンバ名も宣言しない。

```
class Node {
  public:
   Node *left, *right;
   TypeCode code;
   union {
      int i;
      double d;
      char *p;
   };
};
```

ここで，iとdとpは，同じ領域に重複して割り当てられているが，Nodeのメンバとして，これらに直接アクセスすることができる。

```
int i = 0; double d = 0; char *p = 0;
switch( np->code ) {
   case is_charstar:
      p = np->p;
      break;
   case is_double:
      d = np->d;
      break;
```

```
        case is_int:
            i = np->i;
            break;
    }
```

C++において共用体が使われることは，相対的にわずかである。多くの場合，その代わりに第5章で解説する継承機能と仮想関数を使ったほうが，安全かつ便利である。

クラスオブジェクトが占める領域を節約するためのもう1つの手段として，特定のビット数だけを必要とする複数の整数型データメンバに，整数サイズの領域を分割して与えることがある。ある整数のメンバが小さい範囲の値だけを持つことがわかっているとき，**ビットフィールド**を使うことで，クラスオブジェクト中の使用領域を柔軟に制御することができる。ビットフィールドは，以下のように“:”およびフィールドサイズを付加したデータメンバの宣言により指定される。

```
class Node {
  public:
    Node *left, *right;
    unsigned int code: 2;
    unsigned int is_leaf: 1;
    unsigned int is_free: 1;
    // ...
};
```

この例では，codeにはNodeオブジェクト中の2ビットが割り当てられて，4つの値を取り扱うことができる。メンバis_leafとis_freeには各1ビットが割り当てられ，2つの値を扱うフラグとしての役割を果たす。これらのフィールドは，オブジェクト中の隣接した領域にまとめてビット単位で詰め込まれる。

ビットフィールドを使ってオブジェクトの使用領域が小さくなるようにメンバをまとめればプログラマはクラスのサイズを制御することができるのだが，この制御の効果があるかどうかは，C++の処理系をいかにうまく使うかにかかっている。クラスオブジェクトの領域を節約するこの制御をうまく使うには，プログラマは対象となる計算機の型のサイズとアラインメント調整，さらにC++の処理系の動作をよく理解している必要がある。

型の宣言は，クラスの中に入れ子にすることができる。クラススコープの中で，入れ子を使って，ほかのクラスや型の定義，enumのタグや記号値を定義することができる。たとえば，列挙TypeCodeを入れ子宣言を使ってクラスNodeの一部とすることもできる。入れ子のenum値は，クラスのメンバとしてメンバアクセス演算子を使ってアクセスすることもできるし，スコープ決定演算子::を使ってアクセスすることもできる。

```
class Node {
  public:
   Node *left, *right;
   enum TypeCode{ is_int, is_double, is_charstar };
   TypeCode code;
   union {
      int i;
      double d;
      char *p;
   };
} *np;
```

この例では，列挙子の名前はこのクラスに局所的である。これらの名前はほかのスコープの名前と衝突しない。またこれらをprivateに置くことで，これらへのアクセスを制限することもできる。なお，enum値はいずれのNodeオブジェクトからも独立しているので，一般的にメンバアクセス演算子“.”あるいは“->”の代わりにスコープ決定演算子::を使って，これらを参照すべきである。

```
switch( np->code ) {
   case Node::is_charstar:
      p = np->p;
      break;
   case Node::is_double:
      d = np->d;
      break;
   case Node::is_int:
      i = np->i;
      break;
}
```

同様に，クラスの外側の宣言では，入れ子のenum型名はスコープ決定演算子で限定することで示される。

```
Node::TypeCode code  = np->code;
```

一般的にデータメンバはクラス中でprivateにし，特別の権限を持った数個の関数によってのみアクセスされるようにしたほうがよい。この結果，データ構造の実装間の依存関係は制限され，プログラムのモジュール性は増す。この章の以降の節では，アクセス保護も解説することにする。データ隠蔽は，データ抽象表現を構成する基本機能である。データの抽象表現は第4章で扱う。

3.3　メンバ関数

クラス定義の中で，friendと指定されずに宣言されている関数は，そのクラスのメンバ関数である。メンバ関数はクラス内部で定義することもでき，その場合それは暗黙のうちにinline関数であると解釈される。メンバ関数を呼び出すためには，メンバアクセス演算子“.”あるいは“->”が使われる。演算子“.”は，クラスのオブジェクトとともに使われる。

```
String s;
s.print();
```

演算子“->”は，クラスのオブジェクトへのポインタとともに使われる。

```
String *sp = new String;
sp->print();
```

メンバ関数の呼び出しも，非メンバ関数と同様にコンパイル時に検査される。クラスオブジェクトと実引数の型が呼び出すべき関数を決定するために使われ，実引数は関数宣言での仮引数の型にマッチするように型変換される。

メンバ関数は，呼び出しに使われたクラス型オブジェクトに対して演算を行う。メンバ関数には，このオブジェクトへのポインタが暗黙の引数として渡され，関数定義中でこれを“this”という名前で明示的に参照することができる。ただし，メンバ関数中ではメンバの参照においてthisが暗黙のうちに使われるので，メンバを参照するためにこのthisを明示的に使う必要はない。

```
class String {
    char *str;
  public:
    // ...
```

```
    void print() { printf( " %s", str ); }
    // ...
};
```

この例では，関数printは暗黙の引数“String *this”を持ち，この関数でのメンバstrへの参照は，this->strと等価である。printが呼び出されると，呼び出しに使われたオブジェクトのアドレスがthisに設定される。たとえば，s.print()であれば，thisにはsのアドレスが設定される。したがって，このときprintの中ではstrへの参照は(&s)->str，つまりs.strを意味している。呼び出しがsp->print()であれば，thisにはspが設定され，sp->strというメンバアクセスが行われる。

通常のポインタとして宣言された引数に，定数へのポインタを渡すことはできない。これを許すと，関数呼び出し中で非定数ポインタを使ったアクセスによってconstオブジェクトの値を変更できてしまうからである。この制限は，メンバ関数の暗黙のthisポインタにも適用される。

```
const String version("Release 4.0.1");

// ...
version.print();  // エラー！ constに対するprintの呼び出し
```

versionのアドレスは“const String *const”であり，これはprintの引数“String *const this”とは型がマッチしない。しかし実際には，printは対象とするオブジェクトを変更しないので，“const String”にprintを適用できるように宣言を変更しよう。

```
class String {
    char *str;
  public:
    // ...
    void print() const;
    // ...
};
```

メンバ関数宣言の引数並びの後にconstを付けることで，this引数の型が定数へのポインタに変わる。これで，printは“const String *const this”引数を持つようになり，const Stringに適用できるようになる。

各メンバ関数は，論理的にはクラス内部に配置されている。メンバ関数の定義がクラス本体の定義の外側で行われる場合には，クラス名とスコープ演算子によって関数のスコープを指示する必要がある。

```
class X {
   int f();
};

int X::f() { /* etc. */ }
```

メンバ関数はクラス内部に入れ子にされているため，メンバ関数と非メンバ関数でスコープ規則に違いがある。メンバ関数において，識別子の宣言は最初にブロックスコープ，次にクラススコープ，最後にファイルスコープという順序で検索が行われる。このため，メンバの宣言によってファイルスコープの宣言を隠すことができる。次の例の関数X::f()での識別子xは，ファイルスコープの変数xではなく，メンバ変数を参照する。

```
class X {
   int x;
   int f();
};

int x;

int X::f() { return x; }    // this->xを返す
```

局所スコープあるいはクラススコープの識別子の宣言によって隠されたファイルスコープの識別子の宣言を参照する場合には，スコープ演算子を使用する。先の例のX::f()がメンバの代わりにファイルスコープの変数の値を返すように変更するために，ファイルスコープのxを指示するスコープ演算子“::”を使用してみよう。

```
int X::f() { return ::x; } // ファイルスコープのxを返す
```

クラススコープは，メンバ関数を取り巻くブロックに等しい。メンバ関数のブロックスコープ内部での宣言は，ちょうど内側のブロックでの宣言が外側のブロックでの宣言を隠すように，メンバ宣言を隠すことができる。次に示すX::f()は，局所変数の値を返す。

```
int X::f() { int x = 3; return x; }
```

スコープ演算子を使うことで，この例がメンバX::x，あるいはファイルスコープの変数::xを返すように変更することもできる。

3.4 演算子関数

言語定義の演算子をクラス型のオペランドに対しても働くように多重定義することができる。これは，最低でも1個のクラス型あるいはクラスへのリファレンス型の引数を取る利用者定義の演算子関数を与えることで行う。関数が“operator #”(#は多重定義される演算子の記号)という名前を持つことを除くと，演算子関数の宣言と定義の文法は，ほかの関数の宣言と定義と同じ構文である。利用者定義の演算子は，関数名“operator #”を使った関数呼び出しのほかにも，記号#による通常の中置形式の式の構文でも呼び出すことができる。このとき式のオペランドは，関数呼び出しの引数として使われる。

同じ構文で呼び出すため，演算子関数は言語定義版の演算子と同じ数のオペランドを持つ必要がある。多重定義された演算子の優先順序は，対応する組み込みの演算子の優先順序と同じである。演算子関数を意図的に実装しない限り，a+=bとa=a+b，a[b]と*(a+b)，あるいは(&p)->xとp.xの間の同等性といった，言語定義の演算子間の関係が利用者定義の演算子に適用されることはない。演算子関数の結果が，言語定義の演算子の結果と何らかの関係を持つ必要もない。

第1章のクラスcomplexに，いくつかの演算子を追加してみよう。

```
class complex {
    // etc.
    complex operator ++();        // 前置の++
    complex operator +=( complex );
};
```

この演算子++は前置演算子である。演算子++と--の宣言に飾りのint引数を与えることで，後置演算子としての指示を与えることもできる。

```
complex operator ++(int);   // 後置の++
```

飾りの引数は，後置演算子を前置演算子と区別するためだけに使われる。コンパイラは，後置演算子が使われるときにこの引数に仮の値を与える。

利用者定義の演算子+=では，

```
c1 += c2;
```

と

```
c1 = c1 + c2;
```

を等価として定義する必要はないが，利用者が困惑することを考えれば，等価として定義したほうがよいだろう。クラスcomplexは数値型として振る舞うことが期待されているため，+=も組み込みの数値演算子と類似の振舞いをするように実装したほうがよい。

“=”，“()”，“[]”，“->”の4つの演算子はメンバ関数にしか成れないが，これらを除けば，演算子関数は非メンバ関数にもメンバ関数にもなることができる。演算子関数がメンバであれば，暗黙の引数thisが第1オペランドである。クラスStringでは，“operator <”をfriendとして実装していた。

```
class String {
   char *str;
  public:
   friend int operator < ( const String &s1,
                           const String &s2 )
      { return strcmp( s1.str, s2.str ) < 0; }
   // etc.
};
```

これを次のように，メンバとして実装することもできる。

```
class String {
   char *str;
  public:
   int operator < ( const String &s2 ) const
      { return strcmp( str, s2.str ) < 0; }
   // etc.
};
```

演算子関数がメンバの場合，第1オペランドは常に型変換なしでクラス型にマッチするものでなければならない。演算子関数は，言語定義版に類似した振舞いをするように定義されることが多い。言語定義版では，どちらのオペランドに対しても型変換が適用される。このため，演算子関数はfriend関数として実装されることが多い。

演算子関数は変わった名前を持ち，クラス型の引数に働く関数である。呼び出しに際しては中置形式を使う必要はなく，ほかの関数と同じ構文で呼び出すこともできる。非メンバ演算子関数の場合は，すべての引数が引数並びで渡される。

```
class String {
   friend int operator < ( const String &,
                           const String &);
   // etc.
};

void sort( String *a, int n ) {
   // etc.
   if( operator < ( a[i+1], a[i] )) {
      // etc.
```

メンバ演算子関数をこのように呼び出す場合には，メンバアクセス演算子が使われる。

```
class String {
   int operator < ( const String &);
   // etc.
};

void sort( String *a, int n ) {
   // etc.
   if( a[i+1].operator <( a[i] ) ) {
      // etc ...
```

演算子関数を関数呼び出し書式で使うとコードがわかりにくくなってしまうので，この表現はあまり使わないほうがよい。演算子関数は，組み込みの演算子に類似した機能を持つ関数を簡潔に表現するために使われるのが最善である。繰り返すが，演算子関数は，あくまでもコードをわかりやすくするための，純粋に構文上の便法である。

3.5 静的メンバ

staticと宣言されたデータメンバとメンバ関数は，クラスの各オブジェクトからは独立した存在である。staticデータメンバは，クラスが定義されるときに作成され[1)]，どのクラスオブジェクトよりも前から存在する。staticメンバデータとstaticメンバ関数は，クラス型オブジェクト経由ではなく，スコープ決定演算子::を使って参

1) 訳注: 正確にはプログラムの起動時に作成される。

照される。また，これらのstaticメンバは，あらゆる特定のクラスオブジェクトからも独立しており，これらのメンバのアドレスとポインタは，記憶クラスstaticのほかのオブジェクトや関数と同じ通常のアドレスとポインタであり，メンバへのアドレスとポインタではない。

staticメンバの例として，クラスStringのオブジェクトがいくつ作成されたかを記録することを考えよう。作成された数を残すために，Stringのstaticメンバの値をコンストラクタによって増やすことにする。また，このstaticメンバの値を表示するために，staticメンバ関数を使うことにする。

```
class String {
   char *str;
   static int count;
  public:
   String() { count++; str = new char; *str = 0; }
   String( char *s ) {
      count++;
      str = new char[ strlen(s)+1 ];
      strcpy( str, s );
   }
   static void usagereport();
   // etc.
}
```

staticメンバであるcountオブジェクトは，プログラム中にただ1つだけ存在する。このオブジェクトは，Stringクラス本体の外側において，名前がクラススコープであることを除けば，大域データと同じように初期設定される。

```
int String::count = 0;
```

String::countの値は，各Stringオブジェクトが初期設定されるたびに，コンストラクタ本体の中で増やされる。プログラムのある時点でString型オブジェクトがいくつ作成されているかを知るには，正しいアクセス権限を持った関数でString::countを参照するか，あるいは，この情報を表示するために以下のstaticメンバ関数を呼び出せばよい。

```
void String::usagereport() {
   printf( "Report on String Usage:" );
   printf( " %d Strings created\n", count );
}
```

staticメンバ関数は，Stringオブジェクトとメンバアクセス演算子ではなく，スコープ演算子を使って呼び出される。

```
String::usagereport();
```

クラスオブジェクトの特定のオブジェクトを演算するほかのメンバ関数と違って，static メンバ関数は暗黙のthis 引数を持たない。したがって，static 関数usagereport はstatic メンバcount をアクセスすることはできるが，String へのthis ポインタは持たないので，各オブジェクトのstr データにアクセスすることはできない。ほかのメンバ関数と同様に，usagereport はクラススコープの中にあり，static メンバをアクセスするためにスコープ演算子を使う必要はない。

3.6 アクセスの制限とフレンド

クラスメンバは，public(公開，一般的にアクセスできる)またはprivate(非公開，アクセスはメンバ関数とフレンド関数からに限られる)のどちらかになれる。メンバのアクセス属性は，宣言がクラス定義中の"public"というラベルで始まる部分にあるか，あるいは"private"というラベルで始まる部分にあるかによって示される。これらのラベルは，クラス定義中に，何回でも任意の順序で現れることができる。クラス定義は，最初private 部で始まり，別の保護レベルを示すラベルが現れるまで続く。

C++には，このほかにもクラス継承を使う際に意味を持つ第3の保護レベルがある。private またはpublic の指定と同様の方法で，protected(限定公開)メンバを指定することができる。継承を使っていない場合，protected メンバへのアクセスはprivate メンバと同じように制限される。継承とprotected メンバの使用法は第5章で説明する。

クラス定義でフレンド宣言されている関数は，クラスのメンバでないにもかかわらず，クラス型のオブジェクトのprivate メンバをアクセスすることが許されている。クラス自体をフレンド宣言することで，ほかのクラスのすべてのメンバ関数がフレンドであると指定することもできる。

```
class X {
   friend class Y;
   int i;
   void f();
};

class Y {
   int f1( X & );
   void f2( X & );
   // etc.
};
```

この例では，Y::f1とY::f2の内部において，引数“X &”のメンバとして，X型オブジェクトのprivateメンバiとf()にアクセスできる。メンバ関数だけがthisを持っているため，フレンド関数ではメンバアクセス演算子“.”あるいは“->”を使う必要がある。

すでに述べたように，共用体では，特に指示がない限りメンバはpublicである。また，クラス型のもう1つの種類に**構造体**がある。これは標準のメンバアクセスレベルが違うことを除けばクラスとまったく同じものである。構造体の定義では，型定義を示すclassがstructに置き代わる。構造体は，最初public部で始まり，別の保護レベルを示すラベルが現れるまで続く。

```
struct String {
   String();
   String( const char * );
   void print();
   friend int operator < ( const String &s1,
                           const String &s2 );
   // etc.
  private:
   char *str;
};
```

このバージョンのStringでは，メンバ関数はすべてpublicであり，データメンバstrだけがprivateである。

C++のアクセス制御の目的は，クラスのprivateメンバが許可なく使われることを防ぐことであるが，必ずしもpublic部インターフェイスの利用者をクラスのprivate

部から切り離すしているわけではではない。たとえば，privateメンバを追加した結果，クラスオブジェクトのレイアウトが変更されて，publicメンバへアクセスしているコードであっても再コンパイルする必要が生ずることもある。クラスのpublic部は，クラス実装のアクセス可能部分を意味するのであって，実装から切り離されたインターフェイスの宣言ではない。

3.7　初期設定と型変換

クラスがコンストラクタを持っている場合，クラス型のオブジェクトが生成される際には，初期設定のためにコンストラクタが必ず使われる。この章の文字列ソートの例では，引数を取らないコンストラクタがString型の配列要素を初期設定する。オブジェクトを初期設定するために，宣言時にコンストラクタへ引数を与えることもできる。

```
String s1( "hi" );
String s2 = "hi";
```

これらの宣言の初期設定子"hi"は，どちらの場合でも，Stringオブジェクトのs1あるいはs2を初期設定するためにコンストラクタの引数として使われる。

オブジェクトがnew演算子によって生成される場合にも，コンストラクタの引数を与えることができる。

```
String *sp;
sp = new String( "hello" );
```

この例では，spはコンストラクタが引数"hello"を使って初期設定したStringオブジェクトを指すように設定される。クラスオブジェクトの宣言時あるいは生成時に引数が用意されていない場合には，引数を取らないコンストラクタか，あるいはデフォルトの引数値を持つコンストラクタが使用される。

ほかのクラスオブジェクトのメンバとしてのクラスオブジェクトも，コンストラクタによって初期設定される。これらのコンストラクタへの引数は，取り囲んでいるクラスのコンストラクタのメンバ初期設定子並びで与えられる。コンストラクタのメンバ初期設定子並びでは，初期設定子を与えることでメンバに初期値を指定することもできる。定数型あるいはリファレンス型メンバのように初期設定が必要とされるメンバは，

この並びに初期設定子を指定する必要がある。メンバ初期設定子並びは，コンストラクタの定義において，“:” によってコンストラクタの引数並びと区切られる。メンバ初期設定子並びは，メンバ名と，コンストラクタの引数並びないしは初期値を “()” で囲んだものの組の並びで指定される。

```
class Node {
   Node *left, *right;
   const int code;
   String str;
  public:
   Node( int, char * );
   void print() { str.print(); }
   // etc.
};

Node::Node( int c, char *s )
    : code( c ), str( s ) { }
```

この例では，Node のコンストラクタは，引数c の値でメンバcode を初期設定し，第 2 引数s をString のコンストラクタに渡して，Node のメンバstr を初期設定している。このコンストラクタNode の定義を与えたとき，

```
Node *np = new Node( acode, "hello" );
np->print();
```

というコードは，

```
hello
```

を出力する。

メンバ初期設定子並びを使う代わりに，コンストラクタ本体で代入を行うことができるように考えるかもしれない。

```
Node::Node( int c, char *s ) {
   code = c; // エラー: 定数への代入
   str = s;
}
```

ほとんどの非クラス型のデータメンバでは，コンストラクタの最初で値を代入しようが，メンバ初期設定子並びで初期値を指示しようが，両者にたいした違いはない。しかし，const メンバは代入で値を変更することができないので，メンバ初期設定子並

びで値を設定する必要がある。リファレンスメンバも同様である。ここでは，メンバcodeはconstであるから，コンストラクタでの代入ではなくメンバ初期設定子並びの初期設定子によって値を設定する必要がある。

メンバstrはNodeコンストラクタ中の代入で設定してもよいが，この場合strは2回設定されることになる。最初に初期設定によって設定され，次に代入によって設定される。Stringクラス型はコンストラクタを持っているので，

```
class String {
    char *str;
  public:
    String();
    String( const char *s );
    // etc.
};
```

Stringオブジェクトは，生成時に常に初期設定される。何か初期設定子が指定されなければ，strの生成時にはStringのデフォルトコンストラクタが使われる。Nodeコンストラクタ本体での代入は，代入式の右辺を型変換した結果で，strの初期値を置換する。この結果，Stringのコンストラクタが2回呼び出される。最初はstrの初期設定時に呼び出され，次に代入時にsを型変換するために呼び出される。代入と初期設定の意味は同じではない。このため，コンストラクタに対する引数のうちメンバを初期設定するためのものは，const型メンバとリファレンス型メンバに対する初期設定子と同様にメンバ初期設定子並びで指示される必要がある。

コンストラクタは引数からクラスオブジェクトを生成することで，引数をクラス型へと型変換する。これは，コンストラクタは型変換演算子としても使われるということを意味している。このことは，文字列ソートプログラムで，文字列へのポインタを関数呼び出し書式の型変換を使ってStringへと型変換し，文字列配列の要素に代入するという例ですでに示した通りである。

```
a[i] = String( buffer );
```

コンストラクタが引数を1つだけ持つ場合には，キャスト書式の型変換演算によってコンストラクタを実行できる。

```
a[i] = ( String )buffer;
```

一般に，関数呼び出し書式の型変換とキャスト書式の型変換の間には互換性があるが，複数の引数を持つコンストラクタによる型変換は関数呼び出し書式を必要とし，構文的に難しい型指定による型変換はキャスト書式を必要とする。

```
x = X(i,j);            // iとjをXに型変換する
fp = (int (*)())g;     // gを関数ポインタに型変換する
```

型定義(typedef)を使って，関数呼び出し書式のキャストを許すようにして，型の式を簡単にすることもできる。

```
typedef int (*Fp)();
fp = Fp(g);
```

式のオペランドと関数呼び出しの引数は，必要とされる言語定義あるいは利用者定義の型変換演算が利用できれば，正しい型へと自動的に型変換される。利用できる型変換のうちどれを選ぶかの選択に曖昧さがあるとエラーになる。自動的に実行できない型変換を実行させるには，型変換演算を明示的に指示する必要がある。

文字列ソートの例では，暗黙のうちに型変換が適用されるので，“const char *”型の入力バッファをString型に変換するときに型変換演算を明示的に指示する必要はない。“String(const char *)”コンストラクタが利用できるため，右辺の型から左辺の型への利用者定義の型変換が与えられる。このように自動的に適用される型変換を使った代入は，

```
a[i] = buffer;
```

型変換を明示的に使った代入とまったく同じ働きをする。どちらの場合にも，Stringコンストラクタが右辺を型変換するために使われる。

コンストラクタはクラス型への変換を用意する。同様に，型変換演算子関数によって，クラス型からの変換を用意することもできる。このための関数は，常に型変換される側のクラスメンバでなければならない。この関数は，“operator *T*”(*T*は，型変換の結果の型の指定する型名)という名前を持つ。

クラスStringに，Stringをもとの“const char *”に型変換するための変換演算子を追加してみよう。

```
class String {
   char *str;
  public:
   operator const char *();
   // etc.
};

String::operator const char *() {
   return str;
}
```

この宣言では特に戻り型が指定されているわけではないが“String::operator const char *”の戻り型は“const char *”である。ここで，型変換演算子がconstを返す理由は，strメンバによって指される配列が，型変換によって返されたポインタを経由して変更されるのを防止するためである。型変換演算子は，暗黙のうちに使われることもあるし，キャストの際に使われることもある。

```
String msg("Hello World");
const char *cp1, *cp2;

cp1 = (const char *)msg;
cp2 = msg;
printf("%s %s",cp1, cp2);
```

この例では，cp1とcp2の両者には，msgの“const char *”への型変換によって同じ値が設定される。このコードを実行すると以下のものが出力される。

```
Hello World Hello World
```

3.8 クラスメンバへのポインタ

staticデータメンバを除くすべてのクラスメンバは，クラスオブジェクトの構成要素となる。クラスメンバのアドレスは，特定のオブジェクトに対する相対「オフセット」であり，クラスメンバへのポインタ型で表現される。

メンバへのポインタ型を宣言するための型修飾子は“X::*”(Xはクラス名)である。メンバへのポインタを間接参照するためには，演算子“.*”あるいは“->*”を使う。メンバへのポインタを間接参照する演算子は，メンバアクセス演算子と同様に左辺値としてクラス型のオペランドを使う必要がある。

以下に，クラスメンバへのポインタを宣言し，設定して使うだけの例を示す。

```
class Node {
  public:
   int code;
   int num;
   void print();
   void report();
};
```

最初に，Nodeのint型データメンバへのポインタを見てみよう。

```
int Node::*pi;

Node n, *np = new Node;
int i, j;

pi = &Node::code;
i = n.*pi;        // n.codeをアクセスする
j = np->*pi;      // np->codeをアクセスする

pi = &Node::num;
i = n.*pi;        // n.numをアクセスする
j = np->*pi;      // np->numをアクセスする
```

piは，Nodeのint型メンバへのポインタであると宣言されている。最初，piはNode::codeを指すように設定され，次にNode::numを指すように設定されて，ここではNode型オブジェクトのメンバをアクセスするために使われている。

メンバ関数へのポインタ型の指定はすこし複雑である。以下では，typedefを使って，引数を取らずvoidを返すNodeのメンバ関数へのポインタ型にPftypeという名前を与え，このポインタ型の変数としてpfを宣言してみる。

```
typedef void ( Node::*Pftype )();
Pftype pf;

pf = &Node::print;
(n.*pf)();        // n.print()を呼び出す
(np->*pf)();      // np->print()を呼び出す

pf = &Node::report;
(n.*pf)();        // n.report()を呼び出す
(np->*pf)();      // np->report()を呼び出す
```

ポインタpfは複数のNode型のオブジェクトに対して，最初はNode::printを呼

び出すために使われ，次にNode::reportを呼び出すために使われている。“n.*pf”と“np->*pf”を囲んでいる括弧は，呼び出し演算子“()”に間接参照演算子を正しく束縛させるために必要である。メンバへのポインタ型の使い道のほとんどは，こういったメンバ関数の呼び出しである。

メンバ関数へのポインタ型の使用法を紹介するために，ここで再び，追加の第3仮引数の値に従ってリストを昇順または降順に並べ換えるように，文字列ソート関数を拡張してみよう。今回はString型の比較演算子のメンバ版を使用する。

```
class String {
    // 隠された実装
   public:
    // etc.
    int operator < ( String &s );
    int operator > ( String &s );
};
```

関数sortでは，Stringのメンバ関数へのポインタ型の変数compareを宣言する。このメンバ関数は，String型引数を取りint型を返すことにする。変数compareは，選択しようとする関数へのポインタであり，“String::operator >”か“String::operator <”のどちらかが設定される。

```
void sort( String *a, int n, int descending ) {
    int changed;

    typedef int ( String::* Ftype ) ( String );
    Ftype compare = descending ?
          String::operator > :
             String::operator <;
    do {
       changed = 0;
       for( int i = 0; i < n-1; i++ )
          if( ( a[i+1].*compare )( a[i] ) ) {
             // 配列要素を交換する
          }
    } while( changed );
}
```

ポインタを使った関数呼び出し

```
( a[i+1].*compare )( a[i] )
```

は，中置形式の関数呼び出しの

```
a[i+1] > a[i]
```

あるいは，

```
a[i+1] < a[i]
```

のいずれかと等価であるが，どちらであるかはcompareの値によって定まる。

演　習

演習 3-1†　double型引数のビット表現を出力する関数を書け。

演習 3-2†　voidポインタを要素として持つリストクラスをデザインせよ。リストの先頭あるいは末尾に新しい要素を追加する演算，リストの先頭の要素を削除する演算，リストが空かどうかを調べる演算を与えよ。ここで，これらの演算をメンバ関数とするか非メンバ関数とするか，あるいは演算子関数とするか非演算子関数とするか。この決定を行った理由を示せ。

演習 3-3†　レコードの格納と取り出しを文字列キーで行うハッシュ表のクラスを書け。ハッシュ表にレコードを挿入し，探索し，削除するためのメンバ関数をpublic部に用意し，実装の細部をクラスのprivate部に隠せ。

演習 3-4　演習3-3のハッシュ表型を二分木のデータ構造を使って再実装せよ。このとき，public部のインターフェイスには同じものを使用せよ。この結果を，いままで通りハッシュ表型と呼ぶのは正しいだろうか。これが間違っているのであれば，先のハッシュ表型の実装は，本当は何と呼ぶのが正しかったのだろうか。

演習 3-5　先のハッシュ表型に述語論理探索を追加せよ。ここでの述語論理探索とは，発見したレコードが述語論理を満たす場合だけ成功を返すというものである。この演算は検索のための関数lookupを多重定義することで実装すべきか，あるいはデフォルト引数は与えるべきか。

演習 3-6 演習 3-5 の述語論理探索を修正し，true(真)，false(偽)，maybe(たぶん)という結果を返す「曖昧」探索を処理できるようにせよ。true となる要素がない場合だけ，曖昧探索が適用されたときにmaybe となる要素が戻される。

演習 3-7 演習 3-6 のハッシュ表型を修正し，レコードをディスクに格納したり，後で取り出したりできるようにせよ。ここで使った解法を考えてみよう。ハッシュ表型のレコードに対して，実装を簡単にするために行った仮定には何があるか。

演習 3-8† ブロックへの出入りを実行時に追跡するためには，コンストラクタとデストラクタをどのように使用すればよいかを示せ。

演習 3-9 クラスオブジェクトのデータメンバの値は可変である。データメンバは，実行中に新しい値を代入されることもあるし，const データメンバでさえも新しい各オブジェクトごとに初期設定される。この初期設定にはプログラムの実行開始以降に得られた情報が使われることもある。

これに対して，メンバ関数の「値」は固定である。所定の名前と引数型のメンバ関数呼び出しは，常に同じ関数を呼び出す。このことの長所と短所は何か。

実行中にメンバ関数の値(呼び出される関数)を変更するための手段をデザインせよ。C++の組み込みのものと比べたとき，この解法によるメンバ関数の実装の長所と短所は何か。

演習 3-10 演習 3-9 では，クラスの各オブジェクトは，オブジェクト単位で独自の関数を持ち，変更される。所定のクラスのすべてのオブジェクトで同時にメンバ関数の値が変更されるように，実装を修正せよ。

演習 3-11 「読み出し専用」クラスメンバを実現するために，インライン関数を使う手法を示せ。

演習 3-12 以下のクラス定義を考えよう。

```
class CharStar {
   char *cp;
  public:
   char *cp_val1() const
       { return cp; }
```

```
    const char *cp_val2() const
        { return cp; }
    // ...
};
```

cp_val1 はconst メンバ関数にすべきだろうか。また，cp_val2 はどうだろうか。

演習 3-13　メンバ関数を使って，複素数値の遅延評価を実現する手法を示せ。

演習 3-14†　以下の各宣言を文で表現せよ。A，B，C はクラス名とする。

```
int A::*a;
int A::*b();
int *A::*c;
int * A::* const d = 0;
int const * A::* const e = 0;
int A::*B::*f;
int A::**B::*C::**&g = 0;
int A::*(A::*h)(int A::*);
int (A::*i);
int (A::*j)();
```

演習 3-15　演習 3-14 の宣言をtypedef を使ってわかりやすくせよ。

4

データの抽象表現



抽象データ型とは，インターフェイスを通してだけ実装内部へアクセスできるように型の実装の細部を覆い隠してあるデータ型のことである。抽象データ型の特性は，内部構造や実装によってではなく，インターフェイスによって定義される。つまり，場合に応じて同じ抽象データ型を複数の手法で実装することができ，さらに，そうしても，これらを利用するコードには何ら影響を与えないということである。データ型が**抽象的**であるという表現を使っているのは，型の特性をインターフェイスだけで定義し，実装の細部は(抽象的に)ぼんやりとした状態にするからである。

C++のクラスにもデータ抽象表現，つまり型の実装をクラス定義の`private`部に隠し，一般に利用できる演算のインターフェイスのみを公開する機能がある。本章では，C++のクラスを使って実装した抽象データ型の例を示し，データ抽象表現の目的を達成するためのクラスデザインについて議論する。

4.1　データの抽象表現とインターフェイス

第 1 章で示した複素数クラスは，抽象データ型の良い例である。

```
class complex {
    double re, im;
  public:
    friend complex operator +( complex, complex );
    // ...
    friend complex operator /( complex, complex );
    complex( double = 0.0, double = 0.0 );
};
```

利用者から見た型の意味は，抽象インターフェイスによって定義されている。抽象データ型の作成では，インターフェイスのデザインが最も本質的である。型のデザインでは，型の**抽象的な性質**を反映し，実装の変更を許すデザインを行うように留意すべきである。これが「データの抽象表現」と第 3 章の「データの保護」の違いである。いくら不正アクセスから型の実装を保護するためのインターフェイスを使ったところで，型の構造を隠さなければ型の実装に依存するコードが書けてしまう。

たとえば，complex型が2つのdoubleで実装されていることをインターフェイスで公開しているのであれば，complexの利用者は，デカルト座標表現として複素数が実装されていることを仮定してコードを書いてもよいわけである。

```
class complex {
   // ...
   double first() { return re; }
   double second() { return im; }
};
extern complex a;
complex b( a.first(), a.second() );
```

デカルト座標表現として実装されたcomplexであれば，aはbに等しい。実装を極座標表現に変更したとしても，complexは2つのdoubleとして実装できるが，このとき必ずしもaがbに等しいという保証はない。

複素数という型の意味の本質は，算術演算のインターフェイスとほかの数値型からの(あるいはほかの数値型への)型変換によって規定されている。

複素数という抽象概念は自明であり，複素数データ型がどういったインターフェイスを持てばよいかは利用者にも明らかである。複素数は加算，減算，乗算，除算が可能であり，このインターフェイス(性質)は，complex型のオペランドを(引数のように)受け取り，complex型の結果を返す4つの演算子+，-，*，/を多重定義することで実装される。正しいアクセス許可を有する非演算子関数によっても，抽象インターフェイスを実装することはできる。

```
class complex {
   // ...
  public:
   friend complex add( complex, complex );
   // ...
   friend complex div( complex, complex );
   complex( double = 0, double = 0 );
};
```

しかし，複素数の抽象データ型の演算にこれらの関数を使っても，言語があらかじめ提供している数値型と演算子を自然に拡張することにはならない。

```
Z = add( add( R, mul( mul( j, omega ), L ) ),
      div( 1, mul( mul( j, omega ), C ) ) );
```

これと，第 1 章の交流インピーダンスの読みやすい式を比較してみよう。

```
Z = R + j * omega * L + 1/( j * omega * C );
```

クラスcomplexにとっては，多重定義の算術演算子によるインターフェイスのほうが直観的である。しかし，演算子の多重定義には誤用される可能性があるという欠点がある。多重定義演算子の意味を決定するのは実装者であるから，+を「減算」，-を「加算」として実装することもできる(悪意でもない限り，まずありえないだろうが)。

演算子の多重定義が誤用される原因のほとんどは，多重定義演算子の使いすぎによるものである。たとえば，複素数の抽象インターフェイスに指数演算を与えることを考えてみよう。C++には指数演算のための演算子がないので，“^”(排他的論理和)のように，複素数では特に意味を持たない既存の演算子に指数演算の役目を押し付けてみようという誘惑にかられる。これは可能であるが，この種の演算子の多重定義は，バグを持ち込み，読みやすさを害するであろう。この理由の 1 つは，^演算子はすでに指数演算とは違う意味を持っていることにある。このため，たとえcomplex式で^演算子を指数演算の意味で使っていたとしても，排他的論理和あるいは何らかの複素数表現だと解釈されてしまうだろう。

また，もう 1 つの理由として，組み込みの指数演算子を持つ言語のほとんどでは指数演算子は加算演算子よりも優先順序が高いが，実際には，C++の(排他的論理和)演算子 “^” は “+” よりも優先順序が低いということがある。このため，クラスcomplexの利用者は，$-1+e^{i\pi}$という式を “-1 + e^(i*pi)” というコードにするかもしれない。しかし，C++の演算子の多重定義では演算子の既存の優先順序や結合規則を変更することができない。したがって，このコードは$(-1+e)^{i\pi}$と解釈される。この場合，演算子の多重定義をあきらめて，代わりに通常の関数を使用するほうがよい。

```
-1 + pow( e, i*pi );
```

演算子の多重定義は，演算子の既存の優先順序と意味が，演算子の新しい使い道を理解する上で利用者の直観的な助けになるときにだけ使うべきである。

第 3 章で解説したように，2 種類の関数がクラス定義のprivate部へアクセスできる。この 2 種類とは，クラスのメンバ関数とfriendと宣言されたフレンド関数であ

る。先の例では，複素数の算術演算をどうしてメンバ関数ではなくフレンド関数として実装することを選んだのだろうか。

```
class complex {
    // ...
  public:
    // メンバ演算子関数
    complex operator +( complex );    // 二項演算子
    complex operator -( complex );    // 二項演算子
    complex operator -();             // 単項演算子
    // ...
};
```

これは，型が混在する式において，複素数とほかの数値型を相互に型変換できるようにするためである。

第3章で解説したように，コンストラクタも演算子関数と同じように抽象インターフェイスの一部である。このクラスcomplexのコンストラクタには2つの役目がある。1つは，complex型のすべてのオブジェクトを初期設定することであり，もう1つはdouble型からcomplex型への変換である。組み込みの数値型をcomplexへ代入したり，組み込みの数値型でcomplexを初期設定したりするときに，必要に応じて，組み込みの数値型からdouble型への言語定義の型変換と，コンストラクタが暗黙のうちに使われる。

```
complex x = 12.34;    // complex( 12.34, 0 )
x = 12;               // complex( (double)12, 0 )
```

コンストラクタは，関数の仮引数を実引数で初期設定するにあたって，型変換を行う必要があるときにも呼び出される。

演算子関数は，本質的には通常の関数と同じである。このため，演算子関数は中置形式でも，関数呼び出し書式でも呼び出すことができる。ここで，メンバ演算子関数を使ってcomplex式を書いてみよう。

```
complex a, b;
double c, d;
a + b;
a.operator +( b );    // すばらしい...
a + d;
a.operator +( d );    // すばらしい...
c + b;                // エラー!
c.operator +( b );    // エラー!
```

最後のcとbの加算で困ったことが起こった。ここでは，cのクラスのメンバ演算子関数を呼び出そうとしているが，cは非クラス型であるからメンバ関数を持たない。このため，この式はc+bあるいは“c.operator +(b)”のどちらで書かれようが無意味である。このように，複素数の算術演算子をメンバ関数で実装してしまうと，複素数演算子の第1オペランドに非複素数を書く際に，利用者が型変換を明示的に書かなければならなくなる。

複素数の演算インターフェイスの実装では既存の演算子を多重定義し，コンストラクタを使って初期設定と型変換を提供した。この際に，C++の型システムを自然に拡張することに留意した。こうすることで，組み込みの型のように容易かつ自然に使える複素数型を取り込んで，C++の型システムが拡張される。

データの抽象表現は，プログラミング言語の表現能力を特定の問題領域に近づけることを可能にする。たとえば，complexデータ型の例では，複素数の利用者は複素数演算を行うプログラムを簡潔明快に書くことができる。実際，C++言語はこうした問題領域で，開発者の考えと実際に書かれるコードの間の距離を減らすために拡張されている。データの抽象表現の能力は，プログラムデザイン時に概念上の抽象概念を表現するために使われたときに最も成果を発揮する。

複素数は広範囲にわたって利用できるデータ型であり，多数のプログラミング言語では言語定義の型となっている。しかし，C++でも抽象データ型をうまくデザインすることで，多くの専門のアプリケーション領域においてこれらと同様の利点が得られる。

以下では，ソート済みの整数のコレクション(sorted collection)を表現する型を作成する。この種の型を必要とするアプリケーションもいくつかあると思われる。ここではcomplexで紹介したアプローチを使って，型のpublicインターフェイスを定義していくことにする。このクラスには，クラスcomplex同様，コピー演算のためのインターフェイスを用意しないことにしたので，デザインにあたっては高い自由度と義務が同時に与えられる。

```
typedef int Etype;
```

```
class SortedCollection {
  public:
    SortedCollection();
    void insert( Etype );
    void apply( void (*)( Etype ) );
};
```

このクラスが表現する概念は単純である。したがって，インターフェイスも単純である。このクラスには，変数を宣言したときに初期設定を行うコンストラクタSortedCollection，新しい整数をコレクションに挿入する(同時にソートを行う)insert，先頭からのすべての要素に対して指定された関数を呼び出すapplyという3つの演算がある。

ここでは，SortedCollectionのインターフェイスを実装するにあたって，演算子関数は使わないと決めたことに注意されたい。この場合演算子関数は，型の利用者が対応する抽象演算を理解することを逆に妨げてしまうと思われる。たとえば式“a + b”の意図する意味は，aとbが複素数であれば明らかであるが，aがSortedCollectionでbが整数の場合は，意味はわかりにくい。また，aがSortedCollectionでbが関数へのポインタの場合は，意味はとてもわかりにくい。抽象データ型のインターフェイスを実装する場合には，演算子関数よりも非演算子関数のほうが一般的に使われる。

これでインターフェイスは定義されたので，利用者はすぐにでもアプリケーションのデザインとコードの作成を始めることができる。

```
#include "scoll.h"

extern void print( int );
extern int read( int & );

void
printint() {
    // 整数を読み込み，ソートし，出力する
    SortedCollection sc;
    int i;
    while( read( i ) )
        sc.insert( i );
    sc.apply( print );
}
```

この抽象概念には，ほかにも読みやすく効果的と思われるインターフェイスのデザインがいくつかあるだろう。ここでは，そのうちの1つのインターフェイスだけを実

際にクラス定義のpublic 部に与えている。現実のインターフェイスの定義では，抽象概念を利用者に理解させることと，利用者の必要性を反映させることが重要である。

4.2　インターフェイスと実装

利用者がSortedCollectionのインターフェイスでアプリケーションを組んでいる間に，実装者はこの型の第1の実装を仕上げることができる。

```
typedef int Etype;

class SortedCollection {
   Etype ary[ 100 ];
   int free;    // 次の空位置
  public:
   SortedCollection()
     : free( 0 ) {}
   void insert( Etype );
   void apply( void (*)( Etype ) );
};

void
SortedCollection::apply( void (*f)( Etype ) ) {
   for( int i = 0; i != free; i++ )
      f( ary[ i ] );
}

void
SortedCollection::insert( Etype el ) {
   // 試作品のため，検査は省略！
   for( int i = free++; i && ary[ i-1 ] > el; i-- )
      ary[ i ] = ary[ i-1 ];
   ary[ i ] = el;
}
```

この実装にいくつかまずいところがあるのは明らかである。要素の数が多いと，この挿入アルゴリズムは効率が悪い。しかし実際には，効率の悪さが目立つ前に要素数が固定長の配列の範囲を超えてしまい，プログラムを破壊してしまうため，このことが問題視されることはまずない。この実装の長所はコードの作成にわずか数分しかかからないことだけであるが，この結果，さらに優れた実装を提供するための作業を行っている間に，型の利用者は自分のアプリケーションをコンパイルし，デバッグを始めることができる。また，プログラミング計画の初期のデザイン段階では，抽象データ型

のpublic インターフェイスをまだ固定することができないので，こういった素早い実装は便利である。このようにすれば，多額の費用をかけて作成したコードを無駄にすることなく，利用者は型を試したり，インターフェイスの修正を要請したりすることができる。

次に，実装を改良して大きいコレクションを正しく，効率良く処理できるようにしてみよう。

```
typedef int Etype;

class SortedCollection {
  public:
    SortedCollection();
    ~SortedCollection();
    void insert( Etype );
    void apply( void (*)( Etype ) );
  private:
    class Tree {    // 二分木
        Etype el;
        Tree *lchild, *rchild;
        Tree( Etype i )
           : el( i ), lchild( 0 ), rchild( 0 ) {}
        ~Tree()
           { delete lchild; delete rchild; }
        void insert( Etype );
        void apply( void (*)( Etype ) );      // 通りがけ順
        friend class SortedCollection;
    } *root;
    SortedCollection( SortedCollection & ); // 7.3 節参照
    void operator =( SortedCollection & );  // 7.3 節参照
};

inline
SortedCollection::SortedCollection()
   : root( 0 ) {}

inline
SortedCollection::~SortedCollection()
   { delete root; }

inline void
SortedCollection::insert( Etype el ) {
   if( root )
      root->insert( el );
   else
      root = new Tree( el );
}
```

```
inline void
SortedCollection::apply( void (*f)( Etype ) )
   { if( root ) root->apply( f ); }
```

このSortedCollectionの実装は，前の版よりも相当複雑であるため，publicインターフェイスをクラスの先頭に移動し，インライン関数のいくつかをクラスの外側で定義するようにした。このほうが，クラスの利用者は一見して抽象型の必須情報を理解できる。また，この型を実装するのに，private部に入れ子クラスTreeを定義しているが，これは，ほかのクラスがSortedCollectionの実装に依存することを防止するのに役立っている。こういった依存性は，必要に応じて実装を修正することを次第に制限してしまうものである。また，SortedCollectionをSortedCollection::Treeのフレンドとして宣言する必要があるとことに注意されたい。クラスは，自分の中のスコープにあるクラスに対しても特別のアクセス権を持たない。

```
void
SortedCollection::Tree::insert( Etype i ) {
   if( i < el )
      if( lchild )
         lchild->insert( i );
      else
         lchild = new Tree( i );
   else
      if( rchild )
         rchild->insert( i );
      else
         rchild = new Tree( i );
}

void
SortedCollection::Tree::apply( void (*f)( Etype ) ) {
   // Treeの各要素に通りがけ順に関数を適用する
   if( lchild )
      lchild->apply( f );
   f( el );
   if( rchild )
      rchild->apply( f );
}
```

第 2 の実装は，最初のものよりも大きいコレクションを扱うことができ，さらに要素の挿入の効率も良い。これらの点で，新しい実装は最初のものより優れている。しかし，最初のものの制限に遭遇しない限り，型の利用者が 2 つの実装の違いに気づく

ことはない。2つの実装は，同じ抽象型の異なる実装にすぎず，抽象概念の意味はまったく変更されていない。

このことは「補助的な機能」，つまり実装特有の機能がプログラムの動作に効力を十分発揮できないといっているのではない。最初のSortedCollectionの実装では，利用者が100個より多い整数を挿入しないという暗黙の了解がある。しかし，この制限は抽象インターフェイスに表現されていないので，利用者がこれに気付かずこの制約に違反してしまうかもしれない。

現実には，実装特有の「癖」がいつの間にかプログラムによって利用されてしまうという状態も多い(しかし実際にはこうした「癖」を利用すべきではない)。たとえば，このソート済み整数のコレクションを使って，整数のただのコレクション(要素はソートされている必要はない)を作成したとしよう。

```
typedef SortedCollection Collection;
```

利用者が，運悪く，このコレクションがたまたまソートされているにすぎないという事実を利用したコードを書いてしまうかもしれない。この後，ソートを止めることで，要素の挿入の効率を改善すべくCollectionを再実装することもあるだろう。そうすると，実装特有の機能に依存するプログラムを事実上すべて破棄せざるをえない。

こういった理由から，抽象データ型のpublicインターフェイスは型の実装者と利用者間の契約と考えて，これに従うべきである。実装者は，インターフェイスが示す意味を(無用の前提を付加せずに)正しく提供する必要があり，利用者は，publicインターフェイスで明示的に示された意味だけをよりどころとしてプログラミングすべきである。

SortedCollectionの第2の実装において，publicインターフェイスにデストラクタを付加したことに気が付かれただろうか。最初の実装のクラスオブジェクトは，すべてを自分の内部に保持していたため，デストラクタを明示的に用意する必要はなかった。しかし，第2の実装では，コレクションに追加された要素を保持する二分木は，new演算子を使ってクラスオブジェクトの外部に構築される。デストラクタは，SortedCollectionオブジェクトの破棄時に，この外部のオブジェクトの記憶領域を確実に解放するために必要である。

抽象データ型の実装デザインでは，**オブジェクトの生活環**の概念に留意すべきである。オブジェクトの生活環とは，オブジェクトが生成されてから破棄されるまでにたどる，オブジェクトの経歴のことである。オブジェクトは，生成時にコンストラクタによって正しく初期設定され，生存中はメンバ関数とフレンド関数によって常に正しい状態に保たれ，破棄時にデストラクタによって寿命中に割り当てられたあらゆる資源を正しく解放してもらう必要がある。

データの抽象表現の第2の例として，第3章のStringデータ型の修正版を考えることにしよう。

```
class String {
  public:
   String( const char * = "" );
   ~String();
   String( const String & );
   operator const char *() const;
   String &operator =( const String & );
   friend String operator +(  const String &, const String & );
   String &operator +=( const String & );
   friend int operator <( const String &, const String & );
   // このほかの関係演算子と等価演算子...
  private:
   class Rep {
      char *str;
      int refs;    // 参照カウンタ
      Rep( const char* );
      ~Rep();
      friend class String;
   } *r; // -> 参照カウンタによって共有される文字列
};
```

このクラスStringでは，文字列領域をできるだけ共有することにした。このため，文字列領域へのポインタと，このポインタを参照するStringの数(参照カウンタ: reference count)をひとまとめにしたクラスString::Repを作成している。クラスStringは，文字列を直接格納する代わりに，このデータ構造へのポインタを格納している。

最初のコンストラクタは非常に簡単であり，指定された文字列でStringを初期設定する手法を示している(これはまた，単一の引数で呼び出すことができるコンストラクタすべてと同じように，引数の型からクラス型への型変換を示している)。

```
String::String( const char *s )
    : r( new Rep( s ) ) {}
```

クラスString::Repは，自分自身を初期設定する。

```
String::Rep::Rep( const char *s )
    : str( strcpy(new char[strlen(s)+1],s) ), refs(1) {}
```

もう1つのコンストラクタは，ほかのStringで新しいStringを初期設定する。この際に，文字列をコピーする代わりに既存の文字列表現の参照カウンタを1だけ増やし，新しいオブジェクトはこの文字列表現を参照するように設定される。この処理は“String &”引数を1つだけ取るコンストラクタに実装されている。

```
String::String( const String &init )
    : r( init.r ) { r->refs++; }
```

代入の意味自体は変わらないが，String::Repに関しては，代入式の左辺(対象)が新しい値を代入される以前に何かを参照していたということを考慮する必要がある。代入の対象であるStringは古いString::Repをもはや参照しないので，このString::Repの参照カウンタの値を1だけ減らす。さらに，この結果参照カウンタが0になれば，String::Repは削除される。

```
String &
String::operator =( const String &str ) {
    if( this != &str ) {
        if( !--r->refs )
            delete r;
        r = str.r;
        r->refs++;
    }
    return *this;
}
```

組み込みの代入演算子の動作に近づけるため，Stringの代入演算子は代入の対象を変更し，さらにオブジェクトの左辺値(lvalue)を返すようにしてある。このため，a=b=cという連鎖の代入も許される。オブジェクトの左辺値を返すために，自分自身を指すものとしてポインタthisが使われている。また，オブジェクトを自分自体へ代入するという可能性を検査している。自分自体への代入でなければ，String::Repオブジェクトを削除した後，引数strの文字列表現のアドレスをrに代入する。

前節で紹介したcomplexとこのStringでは，代入と初期設定の意味に違いがあることに注意されたい。複素数の実装では，言語定義の代入と初期設定の意味だけで十分であり，ほかの複素数のメンバが単にコピーされればよかった。しかし，Stringの正しい実装では，代入前にStringが参照しているString::Repの参照カウンタを更新する必要がある。このため，クラスStringには演算子関数“operator =”を用意する必要がある。

一般に，ある抽象データ型に代入演算子の定義が必要であれば，初期設定のためのコンストラクタも必要である。逆もまた同じである。String型でどちらかだけを用意することは，両者を省略したのと同じくらい無意味である。これらの演算はどちらも，様々な状況で与えられたクラス型のオブジェクトをコピーする方法を指示している。また，コピーの意味の詳しい解説は，第 7 章で行う。

Stringを削除したり，スコープから抜ける際にも参照カウンタを調整する必要がある。このため，デストラクタの定義も必須である。

```
String::~String()
   { if( !--r->refs ) delete r; }
```

これらに加えて(複素数型と同様に)C++の既存の型システムを自然に拡張するためのインターフェイスとして，いくつかの多重定義演算子が用意される。コンストラクタが文字列からStringへの型変換を提供しているので，演算子+をフレンドとして多重定義するだけで，複素数の算術演算子のように，文字列の連結演算が実現できる。

```
extern char *home_dir, *path, *file;
String home = home_dir;
String fpath = home + "/" + path + "/" + file;
```

String型のインターフェイスでは，さらに，既存の型システムとの関係を完全にするために，Stringから“char *”への型変換演算子も定義されている。この型変換演算子では，文字列の複製のアドレスを返す代わりにprivate部の文字列自体へのポインタを返すように実装した。この実装は効率は良いが，潜在的には危険をはらんでいる。しかし，ここでは定数文字列へのポインタを返しているので，利用者は文字を修正するためにキャストを実行する必要がある。C++のデータ隠蔽機能はアクシデントに対する防護であって，故意の悪用に対するものではないのである。

```
String::operator const char *() const
    { return r->str; }
// ...
char *newfile = fpath;  // fpath.operator const char *()
```

この型変換は，事実上コンストラクタと逆の機能を提供する。クラスStringのコンストラクタは，“char *”あるいはほかのStringからStringを作成する手法を示し，“operator const char *”は，Stringから“char *”を作成する手法を示す。この機能は，“char *”型のオブジェクトに対してすでに書き溜められているコードへのインターフェイスを提供するという意味で非常に重要である。

```
extern FILE *fopen( const char *, const char * );
FILE *fp = fopen( fpath, "r+" );
```

型変換演算子は，コンストラクタと同様に式あるいは初期設定子の中で，暗黙のうちに呼び出される。

しかし，代入演算子の左辺値にこの種の型変換が適用されては困る。つまり，代入では，メンバ関数による，引数の自動的な型変換を抑制する「特性」が必要とされるわけである。このため，Stringの代入演算子は，連結のための多重定義演算子“+”や複素数の算術演算子とは違って，メンバ関数として実装されている。実際，C++では“operator =”はメンバとする必要がある。ただし，ほかの代入演算子にはこの制限はないので注意されたい。

たとえば，“operator +=”がフレンド関数として実装されると，文字列定数にStringを代入することができてしまう。これは，代入式の左辺値の“char *”型が，“const char *”を引数として受け取るStringコンストラクタによって自動的にString型に変換されてしまうからである。

```
"/usr/bin" += pathname;
```

繰り返すが，多重定義演算子の中置形式は表記上の便法であり，関数呼び出しと等価であることを思い出していただきたい。

```
operator +=( "/usr/bin", pathname );
```

したがって，文字列定数への代入は常識的に考えれば誤りであるが，“operator +=”

がStringクラスの非メンバであれば，これは文法的には正しい(意味的には誤りであるが)。ここで，“operator +=”をStringのメンバ関数にすれば，先の代入は次のものと同じこととなる。

```
"/usr/bin".operator +=( pathname );
```

これは無意味であり，文法的にも誤りである。Stringの代入演算子をメンバ関数として実装したため，代入式の左辺値に型変換は適用されず，また文字列定数"/usr/bin"にはメンバ関数がないので，この呼び出しは失敗する。もしStringを“const char *”に代入したり，Stringで“char *”を初期設定したりしたいのであれば，Stringの型変換演算子“operator const char *”によって処理するのがよいだろう。

演算子とコンストラクタと型変換を適切に実装することで，初めて，C++の言語定義の型システムと自在に混在させることができ，型システムを自然に拡張する優れた抽象データ型をデザインすることができるのである。

4.3 制御の抽象表現

クラスSortedCollectionのapply関数は，実際には2つの独立した概念から構成されている。1つはSortedCollectionの要素を保持するデータ構造を最初から最後まで走査することで，もう1つはコレクションの各要素に関数を適用させることである。これらの概念はどちらも**制御抽象**と呼ばれる一般概念として位置付けられる。

適用する演算を与えずにデータ構造の隠れた表現を走査することは滅多にない。しかし，走査を各段階に分解できれば，各段階は興味深い値を生じ，この結果，型の内部構造を隠したままでも利用者は型に関係する値を読み書きできるようになる。この種の制御抽象は**反復子**(イテレータとも呼ばれる)と呼ばれる。

リスト型の実装を考えてみよう。リストには先頭と末尾と中間の要素の列がある。このリストの各要素の値を順番にアクセスできる機構をデザインする。しかし，走査演算をリスト自体に組み込んでしまうと，リストに対して同時に複数の走査を実行することが不可能になるので，これは現実的ではない。

この代案として，リストクラスの中にはデータ構造を組み立てる演算をまとめ，独立した制御抽象クラスの中に走査演算をまとめて，データ構造と走査を明確に分離す

るというものが考えられる。

```
class List {
  public:
   List();
   ~List();
   void append( Etype );
   friend class ListIter;
  private:
   struct Node {
      Node *next;
      Etype el;
   } *head, *tail;
};

class ListIter {
  public:
   ListIter( const List & );   // 先頭を指すように設定
   int done() const;           // 反復終了かどうか
   Etype get() const;          // 現在の要素
   void next();                // 次の要素に移動
  private:
   List::Node *current;
};
```

ここでは，リスト型に対する反復子型(ListIter)をデザインしている。この反復子型は，リスト型がリストという抽象データ構造を実現しているのと同様に，リスト型の要素を最初から最後まで走査するという制御抽象を実現している。リストの要素の走査は，反復子オブジェクトを作成し，これを初期設定する際に走査対象のリストオブジェクトと結び付けることで始める。リスト反復子のpublic インターフェイスはfor，whileといった繰り返し構文中で使われるようにデザインされている。

```
void
print_list( const List &seq ) {
   for( ListIter i = seq; !i.done(); i.next() )
      print( i.get() );
}
```

ここでは，反復子オブジェクトが各呼び出しごとに反復処理の状態を保存しているということが重要である。この状態は反復対象のリストの状態とは独立して管理されているので，同じ１つのリストに対して，複数の反復子を働かせることもできる。これら複数の反復子は，互いに干渉することはない。

```
inline
ListIter::ListIter( const List &ilist )
   : current( ilist.head ) {}

inline int
ListIter::done() const
   { return current == 0; }

inline Etype
ListIter::get() const
   { return current->el; }

inline void
ListIter::next()
   { if( current ) current = current->next; }
```

要するに，反復子はリスト型に「**制御の流れ**」を扱う抽象演算を与えるための拡張キットである。この「制御の流れ」抽象型は，走査目的の抽象データ型と接続することで生成される。この制御抽象型を使うことによって，走査対象の抽象データ型の内部構造の実装から独立した手段で走査を実現できる。

ここでListIterオブジェクトは，ほかのオブジェクトを初期設定したり，ほかのオブジェクトに代入されたりすることにも使うことができるという点にことに注意されたい。

```
void
ripple( const List &list ) {
   for( ListIter i = list; !i.done(); i.next() ) {
      for( ListIter j = i; !j.done(); j.next() )
         print( j.get() );
      print( '\n' );
   }
}
```

このリスト型に関する反復子は非常に簡単である。しかし，さらに多くの役割を実装することもできる。たとえば，反復子を作成する際に，反復処理が終了するまでは新しい要素を挿入できないようにリストをロックする能力を持たせることもできる。もちろん，リストに対する通常動作を定義することも可能である。たとえば，クラスListIterに，引数の**述語論理関数**を満足するリスト要素だけを返す機能を加えたり，直前の要素やリストの先頭へ戻る能力を加えたりすることもできる。また，同じ抽象型に対して多種類の反復子型を作成することもできる。

```
class Employee {
   // ...
  public:
   int isMgr();
   int isTrue()
      { return 1; }
};

extern void print( Employee * );
extern void fire( Employee * );
typedef Employee *Etype;

class List {
   // ...
};

class ListCtl {
  public:
   ListCtl( const List & );
   ~ListCtl();
   typedef int (Employee::*MemPred)() const;
   typedef int (*NonmemPred)(const Employee &);
   Etype operator()();                   // 次の要素に移動
   Etype operator()( MemPred );          // 述語論理を満たす次の要素に移動
   Etype operator()( NonmemPred );       // 述語論理を満たす次の要素に移動
  private:
   // ...
};

void
down_size( List &alist, int all ) {
   ListCtl next = alist;
   Employee *e;
   while( e = next() )
      print( e );
   int (Employee::*criterion)()
      = all ? &Employee::isTrue : &Employee::isMgr;
   ListCtl next_victim = alist;
   while( e = next_victim( criterion ) )
      fire( e );
}
```

制御抽象の背後にある2つの考えは，この章の最初でデータ抽象を一般に議論したときのものと同じである。第1に，制御抽象の利用者は実装の細部に影響されない。これによって「制御の流れ」の基本構造を書こうとする際に，修得する必要がある知識を減らすことができる。さらに型の実装者は，利用者コードに影響を与えずに型の実装

(と型が関係する制御抽象の実装)を変更することができる。

第2に，制御抽象はデータ抽象が型の概観を構築するのと同様に，プログラミング言語の表現能力を問題に関係する制御の概念に近づける。これは，状況が複雑であればあるほど重要であり，これによって，状況が複雑であれば実装の細部で紛失してしまったであろう，意味のある発想を生かすこともできる。

制御抽象とデータ抽象，あるいは抽象データ型と非抽象データ型を分離する線はないし，またあるべきでない。抽象データ型を使うことは，自分の問題ではどういったデータ型を扱ったらよいか決める際に自らの創造力を解放する機会でもある。整数，複素数，文字列，リスト，通信ネットワーク，電話，文法といった概念はすべて(プログラムが関係する限り)，抽象データ型のpublic インターフェイスで表現される抽象的な性質を持っている。データ抽象はこれらの型の新しい実装と，それらを自然に働かせるための新しい制御構造を作成するための招待状でもある。

4.4 総称型

整数のソート済みコレクションは，ある種のアプリケーションのデータ型としては便利であるが，実際には，double あるいはString のソート済みコレクションが必要とされることもあるだろうし，ほかの型のソート済みコレクションのほうが役に立つこともあるだろう。ただし，ここで考えるべき「抽象表現」は，何か特定の型のオブジェクトのソート済みコレクションではなく，ソート済みのコレクションという漠然とした型であることは明らかである。ここでは，SortedCollection の実装をパラメタ対応にし(コレクションの要素型をパラメタで指示できるようにし)，指定された要素型のコレクション型の実体を生成し，オブジェクトを作成することができるようにしよう。このパラメタ対応のSortedCollection の実装は，ソート済みコレクションの総称表現としての役目を果しており，必要に応じて要素型を指定することで，ある特定の要素型のソート済みコレクションの型を新たに生成することができる。利用者は，生成された型のオブジェクトを宣言して使用することができる。

SortedCollection の第2の実装について考えよう。先の例では，Etype 型の要素に対して動作するようにSortedCollection を実装していた。ここでは，typedef で，Etype をString と定義する。この実装では，Etype にはどういっ

た特性が要求されているだろうか。コード全体を眺めると，Etype は演算子=と演算子<を持つ必要があり，またEtype 型は，ほかのEtype 型で初期設定できる必要がある(SortedCollection::apply が呼び出す関数のEtype 型の引数を初期設定するため)ことがわかる。これらを満たす型であれば，この型を単にtypedef で指定するだけで，与えられたEtype のためのSortedCollection を作成できる(以下の例では，この章で前述した，String から“char *”への型変換と，逆の型変換が暗黙のうちに使われていることに注意されたい)。

```
typedef String Etype;
class SortedCollection {
    // ...
};

extern void print( const char * );
extern void censor( char * );
extern char *readstr();

void
printString() {
    // 入力された文をソートして,
    // ごみを取り除いた後, 出力する
    SortedCollection list;
    char *s;
    while( s = readstr() )
        list.insert( s );
    list.apply( censor );
    list.apply( print );
}
```

この手段で，何とか1種類のソート済みコレクションを使うことができる。しかしEtype に2つの型を同時に定義することはできないため，このままでは複数の型を使うことはできない。クラステンプレート機能を使えばこの問題に対処できる。

```
template <class Etype>
class SCollection {
    // ...
  public:
   SCollection();
   ~SCollection();
   void insert( Etype );
   void apply( void (*)( Etype ) );
};
```

第２章で解説した関数テンプレートと同様に，予約語templateとテンプレートパラメタによって，クラステンプレートが導入される。

このクラステンプレートによって，ソート済みコレクションの総称クラスのいくつかの実体を生成し，生成した型のオブジェクトを宣言することができる。ちょうどテンプレートメンバ関数の実体が関数の呼び出し時に暗黙のうちに生成されたように，クラステンプレートの実体は，型名が使われた時に暗黙のうちに生成される。テンプレートクラス名は，クラステンプレート名と<>で囲まれた１つ以上のパラメタから構成される。

```
SCollection<int> ints;
```

この宣言は，intを格納するSCollection型を生成し，生成した型のオブジェクトを作成している。生成したテンプレート型の名前は長いので，typedefを使ってわかりやすい名前を付けることもできる。

```
typedef SCollection<int> SCInt;
SCInt ints;
```

以下のようにして，ソート済みコレクションの総称テンプレートから，まず特定の要素型のためのクラス型を生成し，次に生成したクラス型のオブジェクトを宣言する。

```
#include "scolltmp.h"
#include "String.h"

extern void print( int );
extern void print( String );
extern int read( int & );
extern char *readstr();
extern void censor( String );

void
printint() {
    // ...
    SCollection<int> sc;
    int i;
    while( read( i ) )
        sc.insert( i );
    sc.apply( print );
}
```

```
void
printString() {
    // ...
    SCollection<String> list;
    char *s;
    while( s = readstr() )
        list.insert( s );
    list.apply( censor );
    list.apply( print );
}
```

4.2節で示したSortedCollectionの第2の実装から，SCollectionクラスを作成することにしよう。

```
#include "tree.h"

template <class Etype>
class SCollection {
    Tree<Etype> *root;
  public:
    SCollection() : root( 0 ) {}
    ~SCollection() { delete root; }
    void insert( Etype el ) {
        if( root ) root->insert( el )
        else root = new Tree<Etype>( el ); }
    void apply( void (*f)( Etype ) )
        { if( root ) root->apply( f ); }
};
```

この実装では，もう1つのクラステンプレートTreeを使うことに注意されたい。Treeテンプレートの実装は，SortedCollection::Treeクラスの実装を改良したものである。

```
template <class T>
class Tree {
    T el;
    Tree<T> *lchild, *rchild;
    Tree( T );
    ~Tree();
    void insert( T );
    void apply( void (*)(T) );
    friend class SCollection<T>;
};
```

クラステンプレートのメンバ関数は，非メンバのテンプレート関数と同じようにし

て定義される。

```
template <class T>
Tree<T>::Tree( T i )
   : el( i ), lchild( 0 ), rchild( 0 ) {}

template <class T>
Tree<T>::~Tree()
   { delete lchild; delete rchild; }

template <class T>
void Tree<T>::apply( void (*f)( T ) ) {
   if( lchild )
      lchild->apply( f );
   f( el );
   if( rchild )
      rchild->apply( f );
}
```

同様にして，複数の型パラメタを使う総称型を定義することもできる。

演　習

演習 4-1　クラスcomplexの実装を，実部と虚部の組の代わりに極座標表現を使うように変更せよ。また，これは利用者コードに影響を与えるか。

演習 4-2　複素数から組み込みの算術型への自動的な型変換を与えるために，クラスcomplexにメンバ“operator double()”を定義すべきだろうか。

演習 4-3　新しい整数の型VeryLongとして，longの 4 倍の収納能力のあるものを作成せよ。この型を既存の型システムに組み入れることができるように，演算子と型変換を多重定義せよ。この際に，VeryLongからlongへの型変換を許したほうがよいだろうか。もしこの型変換を禁ずるとすると，if 文あるいはswitch 文の条件値にVeryLong値を許したほうがよいのだろうか。このように決定する理由は何か。また，利用者がこの型に期待するものは何だろうか。彼らが期待するものには何が適切だろうか，また実装者はどのように決定すべきだろうか。

演習 4-4 無限長の有理数を実現する型をデザインせよ。この型を既存の型システムに組み入れることができるように，演算子と型変換を多重定義せよ。

演習 4-5† 整数型のスタックをデザインせよ。このスタックには，どういう演算を与えるべきだろうか。これらの演算の実装はメンバとすべきか非メンバとすべきか。また，演算子関数とすべきか非演算子関数とすべきか。

演習 4-6 演習 4-5 のスタック型を，テンプレートを使った総称スタック型として再実装せよ。また，整数型のスタックを格納するスタック型を生成するために，この総称スタック型を使ってみよ。

演習 4-7 この章のソート済みコレクション型と同じ機能を持つが，格納できる要素数に制限があるソート済みコレクションをデザインし，実装せよ。実装にあたっては，この章のSortedCollectionの第 1 の実装を土台とすることで実現せよ。この際に，オブジェクトの生活環に留意せよ。

演習 4-8 この章のList，ListIter，ListCtl型を，環状リストを使って再実装せよ。この際に，型の抽象インターフェイスを変更することは禁止する。

演習 4-9 この章のList，ListIter，ListCtl型を，パラメタで要素型を指定できるクラステンプレートとして再実装せよ。

演習 4-10† 演習 3-2 のポインタリスト型を修正し，あらゆるポインタ型用の実体が生成できるようにせよ。

演習 4-11 次のインターフェイスを持つ日付の抽象データ型を考えよう。

```
class Date {
  public:
   Date( char * );
   Date( int, int, int );
   const char *str();
   Date &operator ++();
   Date &operator --();
   friend int operator ==( const Date &, const Date & );
   friend int operator !=( const Date &, const Date & );
   friend int operator <( const Date &, const Date & );
   // etc.
};
```

Dateは“17 June 1775”といった日付の文字列表現，あるいは年月日(1775, 6, 17)を表現する3つの整数のいずれかによって初期設定される。型変換演算子は日付の文字列表現へのポインタを返す。演算子++と--は，日付を1日ずつ増やしたり減らしたりする。

日付の内部表現の実装に1つの整数を使うもの，3つの整数を使うもの，文字列を使うものという3つの実装を用意せよ。各実装を相互に比較したとき，各実装の長所は何か。3つの実装全部がただ1つの抽象型を表現していることの長所は何か。

演習4-12†　先のDateクラスのインターフェイスは，以下の書式で使うことを許してしまうことに注意しよう。

```
Date d( 17, 6, 1775 );
// ...
if( d < "2 July 1776" ) {
    // ...
```

文字列によってDateを初期設定できるままで，こういった関係演算や等値演算の式において，文字列からDateへの型変換を禁止するテクニックを考えよ。

演習4-13　物理系に関連する計算のほとんどは，単なるスカラーのオペランドからだけではなく，複数の関連する物理単位のオペランドから構成される。たとえば，交流インピーダンスの計算はオーム，ファラッドといった要素から構成される。こうした計算を行うときに，数値がオペランドである場合と同様に，物理単位の型を用意して代数演算を行うことは検査として有効である。物理単位の計算結果の型は，予想される物理単位の型とマッチするように型変換されなければならない。関係する物理単位を表す整数，浮動小数点数，複素数のデータ型をデザインせよ。計算結果として，正しい数値と物理単位を返すように算術演算子を多重定義せよ。これらの型と組み込みの算術型と複素数型との間の型変換を用意せよ。

演習4-14†　この章のString型に+，+=，<演算子を与えよ。

演習4-15　有向グラフデータ型をデザインし，以下のことを考えよ。内部表現は隠されているか。インターフェイスはできるだけ小さくすっきりしているか。これに反して，インターフェイスは，型の利用者が役に立つ手続きを書ける程度に充実しているか。

演習 4-16　先の有向グラフ抽象データ型で，グラフに対する深さ優先の極大木集合に対し，節の列を走査する制御抽象を書け。

演習 4-17　有限状態オートマトン(FSA)データ型をデザインせよ。この型に対する初期設定子の集合とは何か。この型のオブジェクトを作成するときに検出されるべきエラー状態とは何か。こういう型の使い道には何があるか。1つの表現で，有向グラフとFSAの2つの抽象データ型としての要求を満たすことができるだろうか。1つの表現と2つの表現を比べたときに，特徴に違いがあるだろうか。また，あるとすれば何か。

演習 4-18[†]　この章のソート済みコレクション用の制御抽象をデザインし，実装せよ。実装にあたっては，配列を使ったSortedCollectionの第1の実装を土台とすることで実現せよ。

演習 4-19　演習 4-18 を再実装し，Treeクラスを使ったSortedCollectionの第2の実装に対して動作するようにせよ。

演習 4-20[†]　生成，破棄，要素の挿入，値の検索といった演算を有する二分木型をデザインせよ。また，木の各要素に通りがけ順で関数を適用する走査も与えよ。

演習 4-21[†]　演習 4-20 の二分木型に制御抽象をデザインせよ。この制御抽象は，呼ばれるたびに通りがけ順に次の要素を返すものとする。

演習 4-22　配列の読み書きの際に，添字の範囲の検査を行う配列を総称型としてデザインせよ。

演習 4-23　抽象インターフェイスは無変更で，String型から参照カウンタを取り除くように書き直せ。このとき，コンストラクタ“String(const String &)”は必要だろうか。“operator =”はどうだろうか。

演習 4-24　以下のコードを仮定しよう。

```
String salut = "Hello, world.";
```

```
void
excitement( char *s ) {
    for( ; *s; s++ )
        if( *s == '.' )
            *s = '!';
}
```

ここで，以下の文に割り当てられる意味は何であろうか。

```
String yo = salut + excitement;
String yoyo = salut + " Goodbye, cruel world."
        + excitement + "...";
```

Stringの定義を変更し，この文が構文的にも意味的にも正しくなるようにせよ。

演習 4-25　ヒストグラムデータ型をデザインし，浮動小数点数の引数で添字付けを行う “operator []” を多重定義せよ。

演習 4-26†　辞書用の連想配列(StringからStringへの対応付け)をデザインせよ。次に，配列の添字と要素の両者に型パラメタを与えることで，この連想配列を総称型にせよ。この連想配列の総称型から生成するとして，どういった型が便利だろうか。

演習 4-27†　リンクポインタを 1 つだけ使うような実装を持つ双方向リストをデザインせよ。リストを両方向に走査できる制御抽象を含んだデザインにせよ。

演習 4-28　SortedCollectionの定義を修正し，述語論理関数で添字付けができるようにせよ。添字演算を実行すると，述語論理を満たす添字集合の要素からなる新しいコレクションを作成することとする。

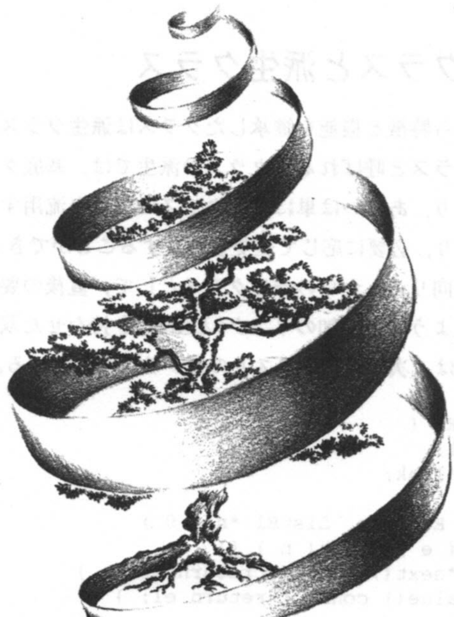
演習 4-29　「延遅実行」型をデザインし，後でオブジェクトにまとめて適用される処理の列の構造を実現せよ。処理の列とオブジェクトを引数として渡すことによって，所定のオブジェクトを延遅実行オブジェクトに結び付ける機能を与えよ。この種の型は何に使われるのだろうか。

演習 4-30　演習 3-2 と各自の解法を審査してみよう。このリスト型の抽象特性のうち，重要であるにもかかわらず問題文での指示に抜けているものは何か。各自の解法においてこれらの特性はどのように実装されているだろうか。この実装のほかに，どういっ

た実装が考えられるだろうか。またほかの解法で再実装した際に，このリストの利用者にどういった影響があるだろうか。

5

継 承



前章で示したcomplex，String，Collection型のように，単独で明確に概念を定義できるのであれば，データ抽象表現は，言語定義の型システムを拡張するためのテクニックとして効果的である。しかし実際には，まったくではないにしろ**ほとんど**望み通りの抽象データ型，いい換えれば，実装あるいは意味が似ている型がすでに存在していることがある。こうした場合には，データ抽象表現の適用範囲を拡げるためのテクニックとして,「継承」を使うことが有効である。

C++の継承は，既存のクラス型の専用版あるいは拡張版として，新しいクラス型を定義するための機構である。本章では，継承の機構と使い道を，コード作成，デザイン，抽象表現という面から解説する。

5.1　基底クラスと派生クラス

ほかのクラスから特徴と機能を継承したクラスは**派生クラス**と呼ばれ，継承のもとのクラスは**基底クラス**と呼ばれる。クラスの派生では，基底クラスの専用版あるいは拡張版を生成したり，あるいは単に基底クラスの実装を流用するために，基底クラスに機能を追加したり，必要に応じて変更したりすることができる。

たとえば，単方向リストからの派生クラスとして，直後の要素と同じように直前の要素を参照できるように，追加のリンク(と演算)を持たせた双方向リストを考えることにしょう。最初は，次の単方向リストの節クラスから始める。

```
class ListEl {
   Etype el;
   ListEl *link;
  public:
   ListEl( Etype e, ListEl *n = 0 )
      : el( e ), link( n ) {}
   ListEl *next() const { return link; }
   Etype value() const { return el; }
};
```

このlistElを使った双方向リストの節は，次のように派生クラスList2Elとして

作成できる。

```
class List2El : public ListEl {
   List2El *link;
  public:
   List2El( Etype, List2El * = 0, List2El * = 0 );
   List2El *previous() const;
};
```

以下の構文は，List2ElがlistElから派生した新しいクラスであることを宣言している。

```
class List2El : public ListEl
```

このときのlistElは，List2Elの基底クラスと呼ばれる。List2Elは，listElのデータメンバとメンバ関数のすべて継承し，さらに直前のリスト節へのリンクと新しいリンクを参照するための関数と新しいコンストラクタを追加している。

メンバ関数previousの実装は，クラスlistElのメンバ関数nextの実装と同様である。

```
List2El *
List2El::previous() const
   { return link; }
```

クラスList2Elは，linkという名前のメンバを2つ持っている。1つは，基底クラスから継承した本来の前方向へのリンクであり，もう1つはList2Elの本体で宣言した後方向へのリンクである。しかし，クラスの派生によってクラススコープが変更されているので，関数previousが返すlinkはList2Elのメンバである。

クラスの派生は，クラススコープの入れ子の階層を構成する。派生クラスのスコープは，関数の中で内側のブロックは外側のブロックの中に入れ子にされるのとほとんど同じようにして，基底クラスのスコープの中に入れ子にされる。したがって，List2Elのメンバ関数中でlinkを参照する場合，コンパイラは最初に派生クラスList2Elのスコープからlinkを探し，発見できなかったときだけ基底クラスlistElのスコープを調べる。同じスペルを持ち，派生クラスメンバで隠されてしまった基底クラスメンバを参照したい場合には，“*base*::*member*”(基底クラス名::メンバ名)という構文を使って，基底クラスのスコープから名前を探し始めるように指示することができる。たとえばpの型が“List2El *”の場合，p->linkEl::linkは，List2Elで宣言

されたデータメンバlinkではなく，基底クラスListElから継承したlinkデータメンバを指す。

関数whatsbeforeでは，継承したメンバ関数と，新規のメンバ関数をどちらも同じ構文で呼び出している。

```
List2El *
whatsbefore( List2El *lst, Etype e ) {
    for( List2El *p = lst; p; p = (List2El *)p->next() )
        if( p->value() == e )
            break;
    if( p )
        return p->previous();
    else
        return 0;
}
```

このコードは，クラススコープの入れ子によってうまく動作する。関数previousの呼び出しだけがクラスList2Elで追加されたメンバ関数を参照し，関数nextとvalueの呼び出しは基底クラスlistElから継承したメンバ関数を参照する。

このようにする代わりに，リストの走査と要素値のアクセスを行うメンバ関数をList2Elに追加することもできたであろう。

```
class List2El : public ListEl {
    List2El *link;
  public:
    List2El( Etype, List2El * = 0, List2El * = 0 );
    ListEl *next() const;
    List2El *previous() const;
    Etype value() const;
};
```

しかしこの案では，運の悪いことに，すぐ問題が生ずる。メンバ関数List2El::nextの実装を考えてみる。

```
ListEl *
List2El::next() const {
    return ListEl::link;  // エラー!
}
```

ここでは基底クラスの前方向へのリンク値を返すことを試みているが，ListEl::linkはprivateメンバであるため，これは不可能である。派生クラスは，フレンド

と宣言されている場合を除いて，基底クラスのprivateメンバに対するアクセス権を一切持っていない。ここで取ることのできる唯一の手段は，基底クラスのprivateメンバをアクセスするためのlistElのpublicインターフェイスを使うことである。

```
return ListEl::next();
```

このコードは動くことは動くが，関数whatsbeforeの実装例で示したように，ほとんど何もしないでも同じ結果が得られるのであれば，こうした努力の価値はほとんどないと思われる。

ここで，List2Elのクラス定義が，以下の宣言で始まっていたことを思い出してほしい。

```
class List2El : public ListEl
```

この文脈での予約語publicは，listElのpublicメンバがList2Elの利用者にもpublicであることを示している。ここで予約語privateを使ったとすると，listElのpublicメンバはList2Elの利用者にはprivateになる。この場合，関数whatsbeforeでのvalueとnextの呼び出しは，すべてprivateメンバへの間違った参照である。ただし，List2Elのメンバ関数とフレンド関数は，listElがpublicまたはprivateの基底クラスであろうとなかろうと，listElのpublicメンバにアクセスできる。

派生クラスのコンストラクタでは，メンバ初期設定子並びによって，明示的に基底クラスを初期設定することもできる。たとえば，以下に示すList2Elのコンストラクタでは，メンバ初期設定子並びによって，基底クラスをあたかも派生クラスのメンバであるかのように初期設定している。基底クラスの初期設定は，メンバ初期設定子並び中で，派生クラスのどのメンバ変数よりも先に初期設定されることを除けば，第3章で述べた通常のメンバ変数の初期設定にとても似ている。ただし，基底クラスに引数なしで呼び出せる(デフォルトの)コンストラクタがあれば，明示的に初期設定する必要はないし，基底クラスにコンストラクタがないのであれば，初期設定の必要すらない(初期設定は暗黙のうちに行われる)。

```
List2El::List2El( Etype e, List2El *fl, List2El *bl )
     : ListEl( e, fl ), link( bl ) {}
```

List2Elコンストラクタのメンバ初期設定子並びで，基底クラスのコンストラクタに渡される引数は，ETYPEと“List2El *”型であるが，listElコンストラクタが期待する型はETYPEと“listEl *”である。“listEl *”を“List2El *”で初期設定することがどうして誤りにならないのだろうか。この理由は，クラスList2ElのオブジェクトはクラスlistElのオブジェクトでもあることにある(この関係はis-a関係と呼ばれる)。このことはクラスの派生宣言で指定されている。さらに，あるクラスBaseがほかのクラスDerivedのpublic基底クラスであれば，DerivedからBaseへの型変換，DerivedへのポインタからBaseへのポインタへ、DerivedへのリファレンスからBaseへのリファレンスへの型変換が，言語定義の型変換として用意される。したがって，多くの状況では「DerivedはBaseでもある」といってもよいわけである。ただし，BaseがDerivedのprivate基底クラスのときには，これらの型変換は存在しないので注意されたい。

このis-a関係と呼ばれる概念は，強力な抽象表現の機構であり，この機構によって，多くの場合，派生クラスを基底クラスとして扱うことができる。たとえば，listElはList2Elのpublic基底クラスであるため，listElから成るリストを期待する関数を，listElとList2Elが複数混在するリスト構造を引数として呼び出すことができる。

```
typedef int Etype;

void
print_list( ListEl *lst ) {
    extern void print( int );
    for( ListEl *p = lst; p; p = p->next() )
        print( p->value() );
}

main() {
    ListEl *lp = new ListEl( 3 );
    lp = new ListEl( 2, lp );
    lp = new ListEl( 1, lp );
    print_list( lp );
```

```
    List2El *l2p = new List2El( 3 );
    l2p = new List2El( 2, l2p );
    l2p = new List2El( 1, l2p );
    print_list( l2p );
}
```

5.2 拡張版と専用版

派生クラスを定義する際に，基底クラスの動作をすこし変更したり，動作を追加したりすることもあるだろう。たとえば，前章では，ファイルのパス名を扱うためにString型を作成した。この際に，Stringから派生したクラスPathnameにパス名に関係する動作を与えることができる。

クラスPathnameに追加したい抽象演算には，パス名が参照するファイルの「オープン」演算のほかにも，パス名を表現する文字列の生成，連結，比較といった演算があるが，これらのほとんどは，すでにString型に与えられている。このため，クラスPathnameはStringの拡張版あるいはStringの専用版と考えることができる。「パス名は…という特性を付加されたStringである」と考えた場合はStringの拡張版であり,「パス名は…という種類のStringである」と考えた場合はStringの専用版である。これらは，いずれも派生クラスを使って表現できる。

```
class Pathname : public String {
  public:
   friend Pathname
       operator +( const Pathname &, const Pathname & );
   Pathname &operator +=( const Pathname & );
   FILE *open();
};
```

この場合，Pathnameのデータメンバはstringから継承したものだけであり，それに加えてパス名に関係する動作がいくつか用意されている。Pathnameは，単にStringというデータ型をほかの視点から見ただけである。

データメンバ，比較演算，“const char *”への型変換はStringから無変更で継承している。しかし連結の実装は，パス名の表現に従ってディレクトリ名をスラッシュ(/)で区切るように変更する必要がある。

```
Pathname
operator +( const Pathname &p1, const Pathname &p2 )
   { return (const String &)p1 + "/" + p2; }
```

Pathname::openは，パス名の文字列表現が参照するファイルをオープンしようとする。

```
FILE *
Pathname::open()
   { return fopen( *this, "r+" ); }
```

ここでのfopenへの実引数*thisは，Stringから継承した型変換“operator const char *”によって，“const char *”へと型変換される。

基底クラスの通常のメンバは，派生クラスへと継承されるが，コンストラクタとデストラクタは継承されない。このPathnameの場合，コンパイラが(Pathnameの基底クラスString部分を初期設定し，後始末するために)コンストラクタとデストラクタが必要であると考え，これらのデフォルト版を自動的に定義する。デフォルト版は，以下に示すように，コンストラクタはメンバ初期設定子並びを持たず，本体が空のインライン関数としてpublic部に定義され，デストラクタは本体が空のインライン関数としてpublic部に定義される。

```
class Pathname : public String {
  public:
   Pathname() {}
   ~Pathname() {}
   // ...
};
```

このコンパイラによって与えれられたコンストラクタは正しく働き，Stringコンストラクタをデフォルト引数で実行する。このため，Pathnameは利用者をびっくりさせる結果も導くこともある。

```
Pathname deadend = "/usr/bin";  // エラー!
Pathname empty;                 // OK...
```

この場合，以下のように，コンストラクタとデストラクタを明示的に与えたほうがよいだろう。

```
class Pathname : public String {
  public:
   Pathname( const char *s = "" ) : String( s ) {}
   Pathname( const String &s ) : String( s ) {}
   ~Pathname() {}
   // ...
};

extern char *home_dir, *path, *file;
Pathname home = home_dir;
Pathname file = home + path + file;
FILE *fp = file.open();
```

ここでは，基底クラスからクラスの動作の大部分を継承し，必要とされる演算だけを基底クラスに追加するという例を示した。これが派生クラスの使い道の中心である。この種の継承は，コードを共有して再利用するためのテクニックとして効果的である。

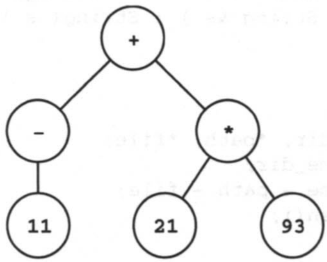
5.3 クラス階層

独立した型としてのデザインが難しい問題であっても，関係する型の集合として考えれば自然に表現できることも多い。たとえば，コンパイラの抽象構文木の節も，複数のプログラミング言語の構成を考慮して表現する必要がある。各種の節型は，共通の核としての特性(その型が「抽象構文木の節」であるための特性)をほかの節型と共有するが，このほかに追加の特性を持ち，これによって，ほかの節型と区別される。データ抽象表現だけを使って，これらの節型すべてを表現しようとすると，すべての節型の特徴を複雑に結合したただ1つの型か，あるいは各節型の共通部分を無視した独立した型の集合のどちらかを作成することになる。しかし，クラス派生を使った継承によって，こうした関連ある節型の集合を作成するにあたって，優れたアプローチを取ることができる。

ここでは，節型の階層を作成するアプローチを取ってみよう。階層の最上位の基底クラスは，すべての派生クラスに共通する構造と機能を提供し，各派生クラスは基底クラスの専用版を与えることにする。

ここで，電卓プログラムのための構文木をデザインしていると仮定しよう。構文木の内部の節が実行すべき演算を表現し，葉はオペランドを表現する。たとえば，この電卓用言語の演算子+，*，単項-の優先順序がC++の優先順序と同じであれば，`-11+21*93`

という式は，以下の構文木で表現できる。



最初に，汎用の節型Nodeを定義することから始める。この節型は，すべての節型の基底クラスの役割を果たす。

```
class Node {
    const int code;
  public:
    enum { PLUS, TIMES, UMINUS, INT };
    Node( int c ) : code( c ) {}
    int eval() const;
};
```

このクラスNodeは，各節オブジェクトの実際の型(節が+，*，単項-，または整数値のどれを表現しているか)を決めるフラグ(code)，コンストラクタ，抽象構文木を評価するためのメンバ関数evalを持つ。

演算子+，*，単項-，構文木の葉の整数値に対応する節型は，Nodeの派生クラスとして作成する。派生クラスのコンストラクタは，基底クラスのコンストラクタを明示的に呼び出し，節の型フラグcodeを正しく設定する。

```
class Plus : public Node {
  public:
    Node *left, *right;
    Plus( Node *l, Node *r )
        : Node( PLUS ), left( l ), right( r ) {}
};

class Times : public Node {
  public:
    Node *left, *right;
```

```
    Times( Node *l, Node *r )
        : Node( TIMES ), left( l ), right( r ) {}
};

class Uminus : public Node {
  public:
    Node *operand;
    Uminus( Node *o )
        : Node( UMINUS ), operand( o ) {}
};

class Int : public Node {
  public:
    int value;
    Int( int v )
        : Node( INT ), value( v ) {}
};
```

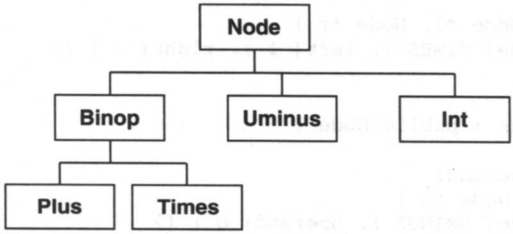
2つの二項演算子(Plus とTimes)の間に重複は多いが，この編成はさほど悪くない。しかし，この電卓用言語にさらに二項演算子を追加する場合には，こうした重複が過剰になり，誤りの潜在的な原因になる。したがって，これらの二項演算子の共通部分を表現するために，もう1つの派生レベルを導入したほうがよい。

```
class Binop : public Node {
  public:
    Node *left, *right;
    Binop( int c, Node *l, Node *r )
        : Node( c ), left( l ), right( r ) {}
};

class Plus : public Binop {
  public:
    Plus( Node *l, Node *r )
        : Binop( PLUS, l, r ) {}
};

class Times : public Binop {
  public:
    Times( Node *l, Node *r )
        : Binop( TIMES, l, r ) {}
};
```

こうして作成されたクラス階層は，節型同士が，継承によってどのように関係しているかを示している。



たとえば，PlusはBinopでもNodeでもあるが，UminusでもTimesでもない。IntはNodeであるが，Binopではない。クラス階層を注意深くデザインすれば，型の間の継承関係は，問題領域において型が表現する概念についての直観的な知識に一致するはずである。つまり，継承機構は，コードの分解と共有を行うだけでなく問題領域での概念構築の助けにもなる。

```
int
Node::eval() const {
    switch( code ) {
    case INT:
        return ((Int *)this)->value;
    case UMINUS:
        return -((Uminus *)this)->operand->eval();
    case PLUS:
        return ((Binop *)this)->left->eval()
                + ((Binop *)this)->right->eval();
    case TIMES:
        return ((Binop *)this)->left->eval()
                * ((Binop *)this)->right->eval();
    default:
        error();
        return 0;
    }
}
```

ここでは，Node::evalを，Node::codeに従って動作を選択しながら，再帰的に部分木を評価するように実装した。ポインタthisを，節の種別フラグが示す派生クラスのオブジェクトにキャストしているのは，部分木を評価するためのメンバ関数evalの呼び出しを目的通りの正しい動作にするためである。キャストの結果，Nodeオブジェクトへのポインタthisは，現在のNode::codeの内容が示す派生クラス型のオブジェクトへのポインタとなる。このポインタのキャストにより，正しい派生クラ

スのメンバ関数を呼び出すことができ，正しい評価を続けることができる。

こうした実装も，限定的に使うのであればさほど悪くはないが，さらに状況が複雑であるときには，再帰の各段階での正しいeval関数の呼び出しを明示的に指示する必要があり，このアプローチではいささか手にあまるものになってくるのが容易にわかる。

このデザインは一応動作し，実際に抽象構文木を構築し，評価することができる。しかし，この実装には大きな欠点がある。ここでの問題は，ここで実装した節型の階層では，基底クラスNodeから派生される型に何があるかが，基底クラス中のフラグcodeの定義によって，固定されていることにある。

この階層に新しい節型を追加する場合の問題は明らかである。たとえば，除算演算子を追加したいとしても，単にBinopから派生するクラス型を宣言するだけでは不十分である。クラスNodeの列挙型にも新しい要素を追加する必要があるし，Node::evalの実装も変更する必要がある。既存の節の階層がライブラリの一部であれば，利用者はこれらの複製を作成し，編集を加えて，独自のバージョンを作成する必要があるだろう。この場合，独自バージョンのライブラリに変更を巧みに施して標準ライブラリとの整合性を取るか，標準ライブラリから離れるかという選択を迫られる。しかし，どちらの場合であっても，これではライブラリを提供するという目的を果たしたことにはならない。

この問題に対しては，特定の節型に固有の情報はすべて，目的の節型の中だけに完全に覆い隠し，階層の上位に位置する節型には下位の型に関する情報を何も記録しないというアプローチが優れている。これを効果的に実現する手段があれば，現在のフラグとキャストを明示的に使ったNode::evalの実装が行っている実行時の節型オブジェクトの型の決定をさらに効果的に行うことができるわけである。これは**仮想関数**(virtual function)によって実現される。

5.4 仮想関数

しばらく抽象構文木の例を離れて，やさしい例を見てみよう。

継承で関係付けられた，果物の種類の階層を扱っていると仮定する。

```
class Fruit {
  public:
   char *identify() { return "Fruit"; }
};

class Apple : public Fruit {
  public:
   char *identify() { return "Apple"; }
;

class Orange : public Fruit {
  public:
   char *identify() { return "Orange"; }
};
```

ここでは，多種類の果物オブジェクトを格納したリストを作成し，各リスト要素に自分の型を決定させることにしよう。ここでは，この章で先に示した単方向リストを使うことにする。

```
typedef Fruit *Etype;
#include "listel.h"

void
print_list( ListEl *lst ) {
   extern void print( char * );
   while( lst ) {
      print( lst->value()->identify() );
      lst = lst->next();
   }
}

main() {
   ListEl *lst = new ListEl( new Fruit );
   lst = new ListEl( new Apple, lst );
   lst = new ListEl( new Orange, lst );
   print_list( lst );
}
```

このプログラムは，リストがOrangeを1個，Appleを1個，Fruitを1個格納しているにもかかわらず，Fruitという単語を3回表示する。ここでlst->value()という式の結果はFruit型へのポインタであるから，これは期待された動作であり，関数Fruit::identifyは文字列Fruitへのポインタを返す。通常，呼び出されるメンバ関数は呼び出しに使われたポインタやリファレンスの型に依存するのであって，ポインタあるいはリファレンスの指す先にある，実際のオブジェクトの型に依存する

のではない。この関数呼び出しは**静的束縛**と呼ばれ，呼び出される関数はプログラムの実行時ではなく，コンパイル時に決定される。

```
Fruit *fp = new Apple;
fp->identify();                    // "Fruit"を返す
((Apple *)fp)->identify(); // "Apple"を返す
Apple *ap = (Apple *)fp;
ap->identify();                    // "Apple"を返す
```

これでは，先程のNode::evalの実装と何ら変わりがない。

しかし，今回の果物プログラムでは，オブジェクトをアクセスするために使うポインタあるいはリファレンスの型ではなく，実際のオブジェクトの型によって正しいidentify関数を決定するようにしたい。このためには仮想関数を使う。

```
class Fruit {
  public:
    virtual char *identify() { return "Fruit"; }
};
```

派生クラスは，仮想関数の機能によって，基底クラスの関数の代替版を定義することができる。たとえば，Fruit::identifyを仮想関数として宣言すれば，Fruitの派生クラスでは，独自版のメンバ関数identifyが用意することができ，関数identifyの呼び出しは実際のオブジェクトの型に従って行われる。つまり，AppleまたはOrangeが，Fruitとしてまとめて扱われていても(Apple型またはOrange型のオブジェクトが，Fruit型のポインタあるいはリファレンスで示されていても)，実際に呼び出されるのは各派生クラスが用意した専用のメンバ関数である。この関数呼び出しは**動的束縛**と呼ばれ，呼び出される関数の本当の版は，コンパイル時にではなくプログラムの実行時に決定される。

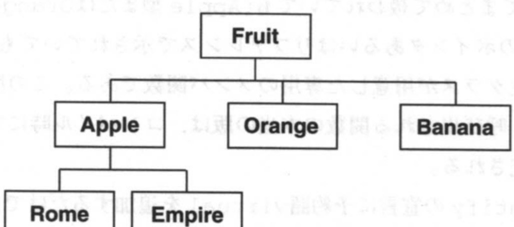
Fruit::identifyの宣言に予約語virtualを追加するだけで，先のプログラムは期待通りに動作し，Orange，Apple，Fruitを表示する。

Apple::identifyとOrange::identifyは，明示的に仮想関数とは宣言されていないが(宣言することもできる)，仮想関数であるということに注意しよう。メンバ関数が仮想関数かどうかを決定する規則は簡単である。仮想関数であると宣言されている関数と，基底クラスに識別記号が同じ(同名で引数の個数と型が同じ)仮想関数が存在する関数は，仮想関数である。この例で，もしApple::identifyの宣言が，

```
char *identify( int = 0 );
```

であれば，この関数はFruit::identifyとは識別記号が違うため，仮想関数ではない。したがって，このプログラムは，Orange，Fruit，Fruitと出力するだろう。また，派生クラスのメンバ関数の識別記号が基底クラスの仮想メンバ関数の識別記号にマッチするのであれば，戻り型も基底クラスのものにマッチする必要がある[1]。このことは，仮想関数が提供する動的束縛の機構が安全であることを保証している。

派生クラスとpublic基底クラスの間のis-a関係と，仮想関数の動的束縛の機構の連携は，抽象表現の手段として有効である。先のprint_list関数は，Fruit型へのポインタのリストを扱うものと考えて定義されている。このprint_listは，クラスFruitを基底クラスとして任意の幅と深さで構築された型の階層を，実装を変更しないで扱うこともできる。たとえば，果物の型は，Fruitから派生したBananaを追加することで階層の幅を拡張したり，あるいはAppleからRomeとEmpireを派生させることで階層の深さを拡張したりすることもできる。各種の派生クラスへのポインタが混在するリストに対して，print_list関数を使うことができるのである。つまり，関係付けられた型の集まりから共通性を抽象的に表現し，この抽象表現に基づいた汎用の手続きを書くことができる。特定の派生クラス型に特有の細部は，各型の中に覆い隠される。



今度は，この概念を使って，抽象構文木の実装を仮想関数を利用するように再デザインしてみることにする。

1) 訳注：新しい標準規格案では，仮想関数の戻り値の制限は緩和された。付録B.4参照。

```
class Node {
  public:
   Node() {}
   virtual ~Node() {}
   virtual int eval() const { error(); return 0; }
};

class Binop : public Node {
  public:
   Node *left, *right;
   ~Binop() { delete left; delete right; }
   Binop( Node *l, Node *r ) : left( l ), right( r ) {}
};

class Plus : public Binop {
  public:
   Plus( Node *l, Node *r ) : Binop( l, r ) {}
   int eval() const
      { return left->eval() + right->eval(); }
};

class Times : public Binop {
  public:
   Times( Node *l, Node *r ) : Binop( l, r ) {}
   int eval() const
      { return left->eval() * right->eval(); }
};

class Uminus : public Node {
   Node *operand;
  public:
   Uminus( Node *o ) : operand( o ) {}
   ~Uminus() { delete operand; }
   int eval() const { return -operand->eval(); }
};

class Int : public Node {
   int value;
  public:
   Int( int v ) : value( v ) {}
   int eval() const { return value; }
};
```

この実装では，メンバ関数evalは，基底クラスNodeで仮想関数として宣言されている。したがって，派生クラスPlus，Times，Uminus，Intのメンバ関数evalも仮想関数である。Binopでは関数evalを宣言していないため，Binopオブジェクトは基底クラスNodeから継承したNode::evalを呼び出すであろう。今度は，抽象構

文木を作成して評価してみよう。

```
int
limited_use() {
   // -11+21*93を評価する
   Node *np =
      new Plus(
         new Uminus(
            new Int( 11 )
         ),
         new Times(
            new Int( 21 ),
            new Int( 93 )
         )
      );
   int result = np->eval();
   delete np;
   return result;
}
```

ここで，npの指す抽象構文木の根には節Plusがあるので，最初の仮想関数evalの呼び出しではPlus::evalが呼び出される。Plus::evalは左右の部分木に対して仮想関数evalを呼び出す。この場合は，Uminus::evalとTimes::evalが呼び出され，同様の呼び出しが繰り返される。各段階で実際に呼び出される関数は，評価を受けている部分木の根にある節の型によって決定される。

クラスNodeは，仮想デストラクタも宣言している。

```
delete np;
```

したがって，この文はnpが指す抽象構文木の根に対する仮想デストラクタを呼び出す。この根にあたる節Plusはデストラクタを明示的に定義していないため，その基底クラスのデストラクタBinop::~Binopがその仕事のすべてを行う。Binop::~Binopは，その左右の部分木に対する仮想デストラクタを呼び出し，続いて同様の呼び出しの繰り返しとなる。この結果，ただ1つのdeleteによって抽象構文木全体が解放されることになる。

コンストラクタは仮想関数になれないことに注意しよう。なぜなら，コンストラクタはオブジェクトを作成するものであるが，仮想関数は呼び出す関数を決定するために既存のオブジェクトを必要とするからである。

仮想関数を使うことで，節特有の演算を特定の節型の宣言中に覆い隠すことができ

る。これにより，既存のコードに影響を与えずに，階層に新しい節型を追加できる。

```
class Div : public Binop {
  public:
   Div( Node *l, Node *r ) : Binop( l, r ) {}
   int eval() const
      { return left->eval() / right->eval(); }
};
```

この例はクラスDivを宣言している。このように，ほかの節型の実装を変更することなく，抽象構文木の節型としてのDivオブジェクトを作成し，評価することができる。

5.5 継承関係のデザイン

Node::evalには2つの役割がある。1つは，同じ識別記号を持つ他の全メンバ関数evalを仮想関数にするための，仮想関数の根としての役割であり，もう1つは，一種の「穴埋め関数」としての役割である。たとえば，Div::evalを宣言し忘れたとする。その結果，節Divに対するevalの呼び出しは，Node::evalへの誤った呼び出しとなる。BinopやNodeのオブジェクトは，実際には作成されて評価されるべきではなく，それらから派生したクラスのオブジェクトだけが，作成されて評価されるべきなのである。階層の根にある穴埋め関数は，この種の誤りを検査する手段として有効である。しかし誤りは，実行時に，しかもその節が評価されたときにしか明らかにならない。

節の階層の利用者がこの種の型のオブジェクトを作成することを禁止したほうがよいし，安全である。これを実現する1つの手段として，NodeとBinopのコンストラクタをprivateにすることがある。しかし，これでは，これらのクラスから派生したクラスであっても，これらのコンストラクタを使うことができない。

```
class Node {
   Node();
  public:
   // ...
};
```

```
class Int : public Node {
    int value;
  public:
    Int( int v ) : value( v ) {
        // エラー！
        // private部のNode::Nodeの暗黙の呼び出し
// ...
```

これと同様の問題がクラスBinopのメンバleftとrightでも発生する。節の階層の利用者が間違ってこれらを使わないように，これらはprivateメンバにすべきであるが，Binopの派生クラス(Plus，Times，Div)から使うことができるようにするために，これらはpublicメンバにする必要がある。

こういった問題を解決するのが**protected**(限定公開)メンバである。protectedメンバは，protectedアクセス指定子とコロン:によってクラス定義に取り入れられる。これは，publicによってpublicメンバが，privateによってprivateメンバが取り入れられるのと同じである。protectedメンバは，privateメンバと同じく，publicメンバのように自由にアクセスされることは許されない。しかし，protectedメンバは，自分が宣言されているクラスの派生クラスのメンバ関数とフレンド関数から，自分が宣言されているクラスのメンバ関数とフレンド関数と同じようにアクセスされてもよい。publicメンバと同じように，非publicの基底クラスのprotectedメンバは派生クラス中ではprivateである。publicの基底クラスのprotectedメンバは派生クラス中でもprotectedメンバである。

```
class Node {
  protected:
    Node() {}
  public:
    virtual ~Node() {}
    virtual int eval() const;
};

class Binop : public Node {
  protected:
    Node *left, *right;
    Binop( Node *l, Node *r ) : left( l ), right( r ) {}
    ~Binop() { delete left; delete right; }
};
```

これによって，NodeやBinopのコンストラクタは，それらのメンバ関数とフレン

ド関数，さらに，それらの派生クラスからだけ呼び出すことができるようになる。その他のコードでは，これらの節型のオブジェクトを生成することはできない。これに加えて，Binopのleftメンバとrightメンバには，Binopから派生した二項演算子型のメンバ関数とフレンド関数だけがアクセスすることができる。

protectedメンバによって与えられる保護は，privateメンバによって与えられるものとは種類が違う。privateメンバはクラス指向であり，それらが宣言されたクラスのメンバ関数とフレンド関数からアクセス可能である。これに対して，protectedメンバはクラス階層指向である。メンバをprotectedと宣言した場合，そのメンバはそのクラスから派生したクラスのオブジェクトで使われることを意図されたものであるということを示す。すでに示したように，クラス派生の目的の1つは，基底クラスの振舞いを調整することである。protectedメンバは，こうした調整のインターフェイスを明示的に表現するためのものである。両者の違いは，基底クラスのprotectedメンバにアクセスしようとするフレンド関数あるいはメンバ関数は，その派生クラス(あるいはさらに，その派生クラスから派生したクラス)のオブジェクトを経由してのみアクセスできるという制限に現れる。

抽象構文木階層の例で，この制限を考えてみよう。Plusのメンバとフレンドは，Plusオブジェクトの基底クラスBinopのleftとrightメンバにアクセスすることができる。PlusのメンバとフレンドはPlusオブジェクト中でleftとrightがどのように「調整」されているかわかっているであろうから，これは適切である。しかし，Plusのメンバあるいはフレンドが，たとえばTimesオブジェクトのBinop::leftあるいはBinop::rightにアクセスすることは，Timesがこれらのメンバを違ったふうに調整していることもあるので，誤りである。

もう1つの例として，前節で示したFruit階層に以下のコードを追加することを考えよう。

```
class Fruit {
  protected:
   int price;
  public:
   void set_price( int p ) { price = p; }
   virtual void print_price() {}
   // ...
};
```

price メンバは単位とも通貨とも結び付いていないので，このままでは意味を成さない。この生の値の正しい解釈はFruit から派生した各クラスで与えられる。

```
class Apple : public Fruit {
  public:
   void print_price()
      { printf( "%d cents (US) / bushel", price ); }
   // ...
};

class Banana : public Fruit {
  public:
   void print_price()
      { printf( "%d centimos (Costa Rica) / crate",
            price ); }
   // ...
};
```

この場合，Apple のメンバとフレンドはBanana オブジェクトのprice を正しく扱うことはできないだろうから，Apple のメンバとフレンドには，Banana オブジェクトを経由して，protected メンバprice へアクセスすることができるという権限は与えられない。

Node 階層にはまだ問題が残っている。1つの問題は安全に関するものである。たとえNode とBinop のコンストラクタがprotected であったとしても，これらのメンバ，フレンドでは，またNode とBinop 型のオブジェクトを作成することができてしまう。もう1つの問題は，インターフェイスと実装に関するものである。基底クラスNode のeval 関数は，たとえ実装が無意味であるとしても，何らかの実装が必要である。これに加えて，派生クラスをデザインする際には，所定の仮想関数をオーバライドする必要があるかどうか，あるいは，その関数の基底クラスの版を継承してよいかどうかが明らかではない。これらの問題は，**純粋仮想関数**(pure virtual function)によって解決される。

純粋仮想関数は，クラス定義の関数宣言に構文“= 0;”を付加することで示される。たとえば，クラスNode のeval メンバを純粋仮想関数にしてみよう。

```
class Node {
    // ...
    virtual int eval() const = 0;
    // ...
};
```

純粋仮想関数には定義は必要ない。純粋仮想関数を持ったクラスは，**抽象基底クラス**と呼ばれ，コンパイラは抽象基底クラスのオブジェクトを作成することを許さない。ただし，抽象基底クラスへのポインタとリファレンスは作成することができる。抽象基底クラスから派生したクラスのうち，1つでもオーバライドされていない純粋仮想関数があるものは，依然として抽象(基底クラス)である。たとえば，Nodeクラスはeval純粋仮想関数を持つので抽象であり，BinopもNodeからevalをオーバライドせずに継承しているので抽象である。

5.6　デザインツールとしての継承

継承とは，取り扱う問題の最も一般的な姿態を型の階層の根に基底クラスとして抽象表現して，この抽象表現だけに従い，モデルとコードを表現するというデザイン法のことである。専用版(この専用版がほかの専用版の抽象表現であることもある)は，基底クラスから派生したクラスとして扱うことができる。

たとえば，抽象構文木の例では，節の一般的な概念から出発し，専用版を定義していくことで節型の階層を拡張した。しかし，本来の抽象表現はしっかりと残っている。

```
void
print_value( Node *np ) {
    extern void print( int );
    print( np->eval() );
}
```

この関数は，Nodeだけを扱い，Nodeから派生した型が持つ複雑さは汎用の手続きから隠されてしまう。

継承をデザインツールとして使うことは，ある意味では，第2章で述べた機能分解という考えによる段階的精製法の使い道に近い。段階的精製法は，問題の手続き上の姿を手続きの階層に分割する。これに対して継承は，問題を表す型の姿を型の階層に分割する。

段階的精製法でプログラムを開発すれば，結果的には常に手続きの階層ができるが，常にトップダウン式で開発が行われるわけではない。しばしば，下位レベルの手続きのコードが先に書かれ，下位の実装が階層の上位にある手続きの構造にどういった影響を与えるかが調べられる。

同様に，多くの型を扱う入り組んだプログラムでは，型が相互にどのように依存しているかをすべて最初に明らかにすることは難しい。表面上はまったく異なるいくつかの型の実装をコード表現したあとで，型相互の共通性が明らかになることもある。このとき，それらのインターフェイスと実装の共通部分を共有するために，継承を使うことができる。多くの場合，この共通性は奥深い抽象表現ができた証拠であり，「発見された」継承階層はデザインの抽象表現として実際に役に立つ。

例として，C++コンパイラのシンボル表の実装を考えてみることにする。わかりやすくするためにすこし簡単にするが，この解法の枠組みは，一般的に，C++コンパイラのシンボル表の開発にも使うことができるし，実際にも使われている。

C++でのスコープの基礎を思い出すことにしよう。C++でのスコープは本質的にブロック構造である。したがって，名前は最初の出現場所から，それが定義されているブロックの終わりまで定義される。内部のブロックである名前が出現すれば，外部のブロックで定義されている同じ識別子を持つすべての名前が隠される。シンボル管理の目的では，ファイル自体も1つのブロックと考えられる。

ブロック構造自体は単純だが，メンバ関数と継承を考えると複雑さは若干増す。メンバ関数のスコープは，その関数が属するクラスのスコープに取り囲まれているということに注意されたい。

```
int i;
class C {
    int i;
    void f() { i = 1; }
};
```

メンバ関数fは，大域変数のiにではなく，クラスメンバのiに値を代入する。次のコードはこれと等価である。

```
int i;
```

```
class C {
   int i;
   void f();
};
void C::f() { i = 1; }
```

コードの表現上，メンバ関数の定義がクラスの内部で行われようが，外部で行われようが，メンバ関数のスコープはそれが属するクラスのスコープに取り囲まれている。

派生クラスのスコープは，その基底クラスのスコープに取り囲まれているということにも注意されたい。たとえば，

```
class B { public: int i; };
int i;
void g() {
   static int i;
   class D1 : public B {
      void f() { i = 1; }
   };
}
class D2 : public B {};
```

というコードでは，クラスBとD1とD2はクラス階層を構成し，クラスD1とD2は，クラスBのスコープに取り囲まれる。特に，D1は，関数gのスコープによって直接取り囲まれるのではなく，最初に基底クラスB，次にgによって取り囲まれることに注意されたい。したがって，D1のメンバ関数fは，関数gあるいはファイルスコープのiにではなく，基底クラスのメンバのiに値を代入する。

シンボル表のデザインに対する最初のアプローチとして，複数の種類のスコープ(大域変数，クラス，関数)をただ1つの固定された構造として管理する代わりに，独立した3つの表で表現することにしよう。各種のシンボル表は，名前の検索と挿入のためのメンバ関数を持つ独立したクラスとして表現される。

```
class Name;

class Gtab {
  public:
   Name *insert( const char * );
   Name *lookup( const char * ) const;
};
```

```
class Ftab {
  public:
   Name *insert( const char * );
   Name *lookup( const char * ) const;
};
```

さらに，派生クラスのスコープは，基底クラスのスコープに取り囲まれているので，クラスのためのシンボル表型Ctabには，基底クラスのシンボル表を参照するためのメンバbase_classが必要である。このメンバは，Ctab::lookupにおいて基底クラスのシンボル表を検索するのに使われる。

```
class Ctab {
   const Ctab *base_class;
  public:
   Name *insert( const char * );
   Name *lookup( const char * ) const;
};
```

挿入関数insertと検索関数lookupの引数は，識別子を表す文字列へのポインタで，戻り値はシンボル表中で名前を表す型Nameへのポインタである。ここでの目的において，Nameは識別子の文字列と関連するいくつかの情報を格納したオブジェクトであることにする。

問題を3つの独立した場合に分割したため，各シンボル表型の抽象的な意味を，各場合ごとに最も適した手法を使って自由に実装することができる。たとえば，クラスのシンボル表にはリスト構造を使い，大域変数のシンボル表には単純なハッシュ表を使い，関数のシンボル表には関数中のブロックスコープを追跡するためにすこし難しいハッシュ表を使うというように，最もうまく構成できる手法を発見して開発を進めることができる。大域変数のシンボル表は，次のように組み立てることができる。

```
class Gtab {
   enum { TSZ = 256 };
   static Name *t[ TSZ ];
   static int hash( const char * );
  public:
   Name *insert( const char * );
   Name *lookup( const char * ) const;
} gtable;
```

これで，与えられたスコープを完全に表現することができる。ここでは，現在の入れ子スコープのスタックを1つのリストで管理する。

```
struct Scope {
   enum { FUNC, CLASS, GLOBAL } type;
   union {
   Ftab *fptr;
   Ctab *cptr;
   Gtab *gptr;
   };
};
List<Scope> scope_list;
```

このアプローチはうまく動作するだろうが，(本来単純であるはずの)挿入と検索のためのコードを非常に複雑にしてしまう。

```
Name *
lookup( const char *id ) {
   Name *n;
   for( ListCtl<Scope> i = scope_list; i; i.next() ) {
      const Scope * const s = i.get();
      switch( s->type ) {
      case Scope::FUNC:
         if( n = s->fptr->lookup( id ) )
            return n;
         break;
      case Scope::CLASS:
         if( n = s->cptr->lookup( id ) )
            return n;
         break;
      case Scope::GLOBAL:
         if( n = s->gptr->lookup( id ) )
            return n;
         break;
      }
   }
   return 0;
}

Name *
insert( const char *id ) {
   const Scope * const curr = scope_list.head();
   if( curr )
      switch( curr->type ) {
      case Scope::FUNC:
         return curr->fptr->insert( id )
      case Scope::CLASS:
         return curr->cptr->insert( id );
      case Scope::GLOBAL:
         return curr->gptr->insert( id );
      }
```

```
    return 0;
  }
```

このデザインでは，これらの関数において汎用の演算で表現したいところでさえもスコープの種類を常に意識する必要が生ずる。

次に，これらの3つのシンボル表型の共通性を考慮し，その共通性を明確にするために継承を使うアプローチが可能である。このアプローチの目的は，本質的には，これらの共通性を基底クラスへと「移す」ことにある。

```
class Tab {
  protected:
   static Tab *curr_tab;    // 現在のスコープ
   Tab *parent;             // 自分を取り囲んでいるスコープ
  public:
   static Name *find( const char * );
   static Name *enter( const char * );
   virtual Name *insert( const char * ) = 0;
   virtual Name *lookup( const char * ) const = 0;
   // ...
};

class Gtab : public Tab { /* ... */ };
class Ctab : public Tab { /* ... */ };
class Ftab : public Tab { /* ... */ };
```

この実装では，シンボル表の階層の根に位置するTab型へのポインタparentによって，自分を取り囲んでいるスコープを明示的に表現している。このTabはGtab，Ctab，Ftabのpublic基底クラスであるから，Tab::parentはあらゆるシンボル表オブジェクトを参照することができる。加えて，現在のスコープを追跡できるように，現在のシンボル表を指す“Tab *”型のクラス型の静的メンバcurr_tabを使うことにする。

これで，C++のシンボル表として使うことができる枠組みが完成した。スコープの状態は常に，関数，クラス，大域変数という3種類のシンボル表全体の結果として決まる。これらのシンボル表間の関係は，各parentポインタ(クラスのシンボル表の場合は，parentポインタと基底クラスポインタによって)で表現される。

たとえば，新しいクラススコープに入るのであれば，クラス本体の最初のところでクラスのシンボル表を新たに作成し，取り囲んでいるスコープと基底クラスを正しく

参照するように初期設定する。クラスが派生クラスであれば，新しいスコープは基底クラスのスコープの中に生じ，base_class は基底クラスのシンボル表を参照するように設定され，parent ポインタはそれを取り囲んでいるクラスないし関数，あるいは大域スコープを表現しているシンボル表を参照するように設定される。新たにデザインしたシンボル表の階層と，それによって使用可能になる仮想関数を利用して書き直すと，汎用の検索関数と挿入関数は非常に単純に表現できる。ここでは，シンボル表の総称型の存在を利用して，これらの汎用演算をクラスTabの静的メンバにすることで，インターフェイスをうまく構築している。

```
Name *
Tab::find( const char *id ) {
    Name *n;
    for( Tab *t = curr_tab; t; t = t->parent )
        if( n = t->lookup( id ) )
            return n;
    return 0;
}

Name *
Tab::enter( const char *id ) {
    { return curr_tab->insert( id ); }
```

最初の3つのシンボル表型を1つの汎用のシンボル表型の特例として表現することで，これらの型を簡単に使うことができるようになっただけでなく，シンボル表の抽象表現を開発する際に，型を考え出す仕事も簡単に行うことができるようになった。

5.7　継承によるインターフェイスの共有

この章で紹介した例の中には，派生クラスによって，すでに十分に使うことができるクラスの拡張版あるいは専用版を作成しているものがある。たとえば，List2El は listEl から，Pathname は String から派生しているが，listEl と String はどちらも実用的で完成された型である。これに対し，シンボル表階層の基底クラスTab と抽象構文木の節階層の Node は非実用的であり，これらの定義はクラス派生によって完成させる必要がある。ある意味では，Tab と Node は，半分実装で半分雛型という不完全型であり，これらから派生するクラスに至って完成される。

さらに突き詰めていくと，あるクラスからの派生クラスが特定のインターフェイス規

約を満たしていることを保証するために，基底クラスを定義するという概念に行きつくことができる。この場合，派生クラスは基底クラスの拡張版あるいは専用版ではなく，基底クラスが定義しているインターフェイス規約の正しい実現と考えるほうがよい。

オペレーティングシステムのカーネルとデバイスドライバ間のインターフェイスを定義する仕事を考えよう。所定のデバイスに対してopen，close，read，writeといったシステムコールを発行するために，インターフェイスとして，汎用のデバイスドライバの振舞い(動作)をまとめたクラス型を実装することにする。

```
class Device {
  protected:
   Device();
  public:
   virtual ~Device();
   virtual int open( char *, int, int );
   virtual int close( int );
   virtual int read( int, char *, unsigned ) = 0;
   virtual int write( int, char *, unsigned ) = 0;
   virtual int ioctl( int, int ... ) = 0;
};
```

デバイスドライバの基底クラス型のメンバ関数には，派生クラスでこれらの関数が未定義であった場合のために，以下のようにデフォルトの動作を実装するか，

```
int
Device::open( char *path, int oflag, int mode )
    { return nulldev(); }
```

あるいは，この演算が純粋仮想関数であると宣言することで，有意のデフォルト動作が不在であることを指定する。

以前に，汎用の基底クラス型Nodeを使って抽象構文木のあらゆる節を評価したり，汎用の基底クラス型Tabを使って各種のシンボル表をアクセスしたりした。同様に，この汎用のデバイスドライバ型を使うことで，カーネルはすべてのデバイスを汎用的に扱うことができるようにする。

この抽象基底クラスから派生させることで，特定の種類のデバイスに対応するデバイスドライバ型を作成する。

```
class Terminal : public Device {
  public:
   Terminal();
   ~Terminal();
   int open( char *, int, int );
   int close( int );
   int read( int, char *, unsigned );
   int write( int, char *, unsigned );
   int ioctl( int, int ... );
};
```

インターフェイスのうち，派生デバイス型の中で明示的に再定義されない部分には，基底クラスで用意された関数によるデフォルトのインターフェイスが使われる。

```
class MemMappedIo : public Device {
  public:
   int read( int, char *, unsigned );
   int write( int, char *, unsigned );
   int ioctl( int, int ... );
};
```

MemMappedIo デバイス型では，純粋仮想関数の定義だけを与えている。これが，MemMappedIo オブジェクトを作成するにあたって最低限要求されるものである。MemMappedIoデバイス型は，Deviceからほかのメンバ関数(デフォルトの実装を持つ仮想関数も一緒に)を継承し，コンパイラは，継承時にデフォルトのコンストラクタとデストラクタを生成する。

5.8 多重継承

派生クラスは複数の基底クラスを持つこともできる。2つ以上の基底クラスを使う継承は**多重継承**(multiple inheritance)と呼ばれる。多重継承は，**単一継承**(ただ1つの基底クラスを使ったもの)ほど一般的に使われるわけではないが，2つ以上のクラス型の特性を兼ね備えたクラスを必要とする場合に，多重継承は便利である。

たとえば，何らかの状態を監視し，その状態をスクリーンに表示する型を作成したいとする。ここでは，移り変わる値を表示する抽象データ型Dialといったビットマップグラフィックスライブラリを使うことができることにする。

```
class Dial {
  public:
   Dial( const char *label, double low, double high );
   ~Dial();
   void set_display_value( double );
  private:
   // ...
};
```

コンストラクタDialは，第1引数に目盛りの名前，第2と第3の引数に測定範囲を受け取って，目盛りを表示する。作成された後，Dialオブジェクトは，Dial::set_display_valueに引数として最後に渡された値を表示し続ける。

Samplerという一定時間ごとに何らかの検査を行う型も用意されていることにする。

```
class Sampler {
  protected:
   Sampler( double frequency );
   virtual void sample() = 0;
  public:
   virtual ~Sampler();
  private:
   // ...
};
```

Samplerオブジェクトの機能は，frequency秒ごとに仮想関数sampleを呼び出すことだけである。ここでのfrequencyはSamplerコンストラクタの引数として与えられる。

Monitor型は，DialとSamplerの両者の性質を持つ。ここでは多重継承を使って，2つのクラスからの継承を表現しよう。

```
class Monitor : public Sampler, public Dial {
   void sample();
  protected:
   Monitor( char *lab, double l, double h, double f )
       : Dial( lab, l, h ), Sampler( f ) {}
   virtual double get_display_value() = 0;
};
```

以下の構文は，MonitorがDialとSamplerの両者から派生した新しいクラス型であることを宣言している。

```
class Monitor : public Sampler, public Dial
```

複数の基底クラスを使った多重継承は，ただ１つの基底クラスから継承する単一継承を自然に拡張したものである。複数のクラス名を基底クラス並びにコンマで区切って書くことができるが，同じ名前が２回現れてはならない。

コンストラクタMonitorでは，メンバ初期設定子並びを使って基底クラスのコンストラクタを明示的に呼び出している。基底クラスの初期設定は，メンバ初期設定子並びで指定された順序には関係なく，基底クラスがクラス定義の基底クラス並びに現れた順序で行われる。また，先に述べたように，たとえメンバの初期設定子並びでメンバ変数名が基底クラス名より前に現れたとしても，基底クラスは常にメンバ変数の前に初期設定される。

クラスMonitorのスコープは，２つある基底クラスの**両方の**スコープの中に入れ子にされている。このためMonitor::sampleは，基底クラスの純粋仮想関数Sampler::sampleをオーバライドする仮想関数である。またMonitor::sample中でのset_display_valueへの参照は，Dial::set_display_valueを参照する。Monitor::sampleの実装は，Samplerのサンプリング機能にDialの表示機能を接続する役目を果す。

```
void
Monitor::sample()
   { set_display_value( get_display_value() ); }
```

この関数は，純粋仮想関数get_diaplay_valueを呼び出し，戻り値をDialから継承した非仮想関数set_display_value関数へ渡す。

多重定義による「多重入れ子」は，単一継承では起こらない曖昧さの原因になることがある。この問題をいくつか例示するために，前に例としてあげた果物(Fruit)階層の根に，木(Tree)を加えたものを考えてみる。

```
class Fruit {
  public:
   virtual char *identify() { return "Fruit"; }
};

class Tree {
  public:
   virtual char *identify() { return "Tree"; }
};
```

いくつかの型はFruitとTreeの両者の性質を持つ。

```
class Apple : public Fruit, public Tree {};

Apple *ap = new Apple;
ap->identify();    // エラー!
```

関数呼び出しが曖昧であるため，このコードはコンパイルできない。クラスAppleのスコープはFruitとTreeの両者のスコープに取り囲まれているため，identifyの呼び出しはFruit::identifyを呼び出すのか，それともTree::identifyを呼び出すのかが判断できない。この場合，スコープ演算子

```
ap->Fruit::identify();
```

を使うか，あるいはキャスト

```
((Fruit *)ap)->identify();
```

を使うかのどちらかによって，参照する基底クラスを明らかにして，呼び出しの曖昧さを解消することができる。さらによい解決法は，クラスAppleに，要求された動作を行う関数identifyを定義することである。この関数は，Fruit::identfyとTree::identfyの両者をオーバライドすることになる。

```
class Apple : public Fruit, public Tree {
  public:
   char *identify() { return "Apple"; }
};
// ...
ap->identify();    // 曖昧でなくなった
```

多重継承を使うときでも，派生クラスと複数の基底クラスの間の関係はすべて，単一継承のときと同じように保たれる。したがって，Apple::identifyはTree::identifyとFruit::identifyの両者を再定義する仮想関数となる。さらに，FruitとTreeはどちらもAppleのpublic基底クラスであるから，AppleからFruitへの型変換と，AppleからTreeへの型変換の2つが言語定義の型変換として与えられる。

```
Apple a;
Tree &t = a;    // OK: AppleはTreeでもある
Fruit *f = &a; // OK: AppleはFruitでもある
```

```
*f = t;          // エラー! TreeはFruitではない
 a = t;          // エラー! FruitはAppleではない
```

前に述べた抽象構文木の節階層での型変換とこれを比較してみてほしい。

Monitorクラスの話に戻る。新しい型を作成するために，多重継承を使ってDialとSamplerの2つの独立した機能を結び付けてみた。型を結び付けるという多重継承の使い道は，とても重要である。

クラスMonitorから，さらに専用のモニタ型を派生させることができる。Monitorから派生した型では，コンストラクタMonitorを経由して，初期値をDialとSamplerのコンストラクタに渡し，監視されている値を与える純粋仮想関数get_display_valueをオーバライドする必要がある。

```
class MemUsage : public Monitor {
   char *start;
   char *max;
  public:
   double get_display_value()
      { return (current_top()-start)/(max-start); }
   MemUsage() : Monitor( "Memory Usage", 0, 100, 0.1 ),
      start( current_top() ), max( get_limit() ) {}
};
```

MemUsageオブジェクトは，自分の属するプロセスが使うことのできるメモリの消費率を監視する。仮想関数get_display_valueは，MemUsageオブジェクトが作成されてからどれくらいメモリが消費されたかを返す。システム関数current_topは，このプロセスが使用できるメモリの先頭アドレスを与える。この値は，MemUsage中の計算で必要である。コンストラクタは，監視目盛りに“Memory Usage”というラベルを貼り付け，表示範囲を0.0から100.0の間に設定し，検査間隔を10回毎秒に決定する。使用可能メモリの消費率を計算するために，毎回current_topを使用していることに注意してほしい。システム関数get_limitは，このプロセスが使用できるメモリの最大アドレスを与える。

演算子の多重定義と同じで，多重継承は，過度に乱用されることの多い機能である。たとえば，Monitor型は，Dialをメンバとして，Samplerから継承することで単一継承として実装することもできただろう。しかし，この場合，DialとMonitorの

間にはis-a関係は存在しない。こうした関係が抽象デザインを理解するにあたって重要であるとか，is-a関係による自動的な型変換を頻繁に使う(Minitorは頻繁にDialとして使われる)という場合には，多重継承を使うのが適切である。さもなれば，単一継承を使って単純に実装したほうがよい。

多重継承の一番ありふれた正しい使用法は，「生来の」単一継承階層の一部であるクラスに，さらに1つ以上のインターフェイスを継承するために多重継承を使うというものである。

たとえば，Monitor型の単一継承版が，コントローラサブシステムによって制定されたプロトコルに返答する必要があることにしよう。プロトコルは次のようにクラス中にまとめることができる。

```
class Protocol {
  protected:
   Protocol();
  public:
   virtual ~Protocol();
   virtual void activate() = 0;
   virtual void deActivate() = 0;
   virtual void reSync() = 0;
};
```

コントローラサブシステムが扱うことのできるのは，Protocolオブジェクトだけであるから，Monitor型はProtocol型である必要もある。したがって，MonitorとProtocolの間にはis-a関係が存在する必要がある。

```
class Monitor : public Sampler, public Protocol {
   // ...
   void activate();
   void deActivate();
   void reSync();
   // ...
};
```

Monitorは，コントローラサブシステムからのメッセージに正しくMonitor流に返答するように，Protocolインターフェイス仕様の純粋仮想関数をオーバライドする。コントローラサブシステムはProtocolクラスだけを扱い，それから派生した型のオブジェクトの存在を特に知る必要はない。

5.9 仮想基底クラス

Monitor型では，2つのまったく独立したクラスを結び付けるために，多重継承を使った。この多重継承は，すこしばかり関連のあるクラスを結ぶ付けるためにも使うこともできる。既存の2つのデバイス型から性質を受け継ぐ，新しいデバイス型を作成したいとする。

```
class MonitoredDevice :
   public Device, public Monitor {
   // ...
};

class NetworkDevice :
   public Device, public Protocol {
   // ...
};
```

MonitoredDeviceはデバイスを通過するデータ伝送速度を監視するものであり，NetworkDeviceは所定のネットワークプロトコルに従ったネットワークドライバボードへのインターフェイスである。ここで，ネットワーク伝送速度を監視する新しいデバイス型を作成してみる。実装の試みとして，次のような「寄せ集めの」継承関係を考えることもできる。

```
class MonitoredNetworkDevice :
   public Device, public Monitor, public Protocol {
   // ...
};
```

しかしこれでは，is-a関係を利用して，新しい型MonitoredNetworkDeviceをNetworkDeviceあるいはMonitoredDeviceとして使うことはできない。

新しい型を既存の2つのデバイス型MonitoredDeviceとNetworkDeviceから派生させる試みとして次の実装が考えられる。しかし，この実装では，結果として階層に2つのDevice型が入ってしまう。

```
class MonitoredNetworkDevice :
   public MonitoredDevice, public NetworkDevice {
   // ...
};
```

この結果が必要とされることもあるが，ここで作成しようとしている新しいデバイ

ス型とは相性が悪い。新しい型をデバイスとして使おうとした時点で，すぐに曖昧さによる誤りが引き起こされ，参照先のオブジェクトのデバイス型をいちいち指定する必要が生ずる。

```
MonitoredNetworkDevice iso;
iso.open( "/dev/iso", O_RDWR, 0 );
        // エラー！ 曖昧
iso.Device::open( "/dev/iso", O_RDWR, 0 );
        // エラー！ まだ曖昧
iso.NetworkDevice::open( "/dev/iso", O_RDWR, 0 );
        // OK: NetworkDeviceのデバイス
iso.MonitoredDevice::open( "/dev/iso", O_RDWR, 0 );
        // OK: MonitoredDeviceのデバイス
```

さらに本質的な問題として，ここではただ1つの物理デバイスを扱いたいにもかかわらず，MonitoredNetworkDevice型が2つの物理デバイスを表現していることがある。これは必ず難事を引き起す。たとえば，MonitoredNetworkDeviceオブジェクトを生成する際には，コンストラクタDeviceが2回呼び出されてしまう。このため，オペレーティングシステムのカーネルがデバイスの演算にDevice型のインターフェイスだけを使っているとしたら，デバイスが1つしかないのに2つあるという結論を下してしまうだろう。

ここでは，MonitoredDeviceとNetworkDeviceの両者から派生させたにもかかわらず，1つのデバイスだけを扱うクラスを作成しよう。これを実現するために，仮想基底クラスを使う。

```
class MonitoredDevice :
   public virtual Device, public Monitor {
   // ...
};

class NetworkDevice :
   public virtual Device, public Protocol {
   // ...
};

class MonitoredNetworkDevice :
   public MonitoredDevice, public NetworkDevice {
   // ...
};
```

基底クラスの仮想宣言は，指定された基底クラスが(それが複数あるように宣言されても)実質的には 1 個だけのオブジェクトを共有することを意味する。この例では，Device はどちらも仮想基底クラスとして宣言されているため，クラス MonitoredNetworkDevice のオブジェクトにはDevice が 1 個しか存在しない。Device の存在がただ 1 つだけになったので，そのメンバへの参照にもう曖昧さは存在しない。

```
iso.open( "/dev/iso", O_RDWR, 0 );   // 曖昧ではない
```

同様に，初期設定すべきDevice も 1 つだけなので，コンストラクタDevice は MonitoredNetworkDevice オブジェクトから 1 回だけ呼び出される。

仮想基底クラスを使うにあたって，気を付けるべき巧妙な仕組みがいくつかある。仮想基底クラスの初期設定はほかの基底クラスの前に行われ，初期設定は，**最派生クラス**(most derived class)の正しいコンストラクタによって行われる。この最派生クラスとは，基底クラスである部分オブジェクトのどれかのクラスではなく，オブジェクト全体のクラスのことである。たとえば，MinotoredNetworkDevice 中のDevice 仮想基底クラスは，NetworkDevice あるいはMinotoredDevice のコンストラクタではなく，MinotoredNetworkDevice のコンストラクタで初期設定される。もちろん，NetworkDevice のオブジェクトでは，初期設定はNetworkDevice のコンストラクタによって行われる。

演　習

演習 5-1　ListEl は，どうして，リストクラスではなく，リスト要素クラスと呼ばれるのだろうか。

演習 5-2　以下の各問題領域を考えよう: 銀行口座，オンラインドキュメント，ソートアルゴリズム，動物園の動物達，レキシカルアナライザが返すトークン，GUI インターフェイスプロトコル，従業員のデータベースレコード。これら各問題領域がどのように系統立てられているかという抽象概念を反映した階層を考え，抽象表現の各レベルごとにどういった演算を関係付けるべきかを決定せよ。さらにこの階層を，クラス派生，非仮想関数，仮想関数，純粋仮想関数を正しく使うことで，C++ 上に実装せよ。

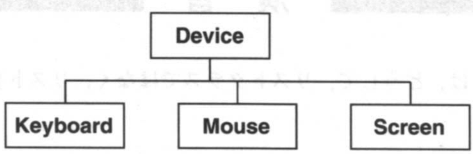
演習 5-3 演習 4-17と同じように，グラフと有限状態オートマトンの抽象データ型について考えてみよう。どちらか一方を他方から派生させることができるか。それらは同じ基底クラスを共有できるか。コードの共有だけを行う継承を使って，それぞれの性質を記述することができるだろうか，またグラフと有限状態オートマトンの間に概念的な共通性はあるだろうか。

演習 5-4† 電卓用言語の抽象構文木の節の階層に，識別子節型と代入節型を追加してみよう。これを行うためにほかの節型の実装を変更しなければならないだろうか。できあがった節の階層を使って，対話的な電卓プログラムを完成させよ。

演習 5-5 `Pathname` 型での “`operator +`”(文字列結合演算子)の実装において，2つの “`Pathname &`” 引数を “`String &`” にキャストしなければ，どのようなことが起こるか。**Hint:** キャストがあれば “`operator +`” のどちらのバージョンが呼ばれ，キャストがなければどちらのバージョンが呼ばれるだろうか。

演習 5-6 `String` を基底クラスとしてではなく，メンバとして使うことで，`Pathname` 型を実装せよ。これら 2 つの `Pathname` の実装の動作上の違いは何だろうか。この場合，継承が `Pathname` から `String` への自動的な型変換を得る目的だけで使われているとしたら，継承の代わりに型変換演算子を使ったほうが適切だろうか。

演習 5-7 次の継承階層を持つ端末デバイスに対する既存のハードウェアインターフェイスライブラリがあったとしよう。



ライブラリは修正できないが，階層にタブレットとライトペン型を追加しなければならなかったとする。どこに，どのようにそれらを追加すればよいだろうか。ライブラリを修正できたとすれば，今度はその型をどのように追加すればよいだろうか。また，将来的な新しい型の導入を容易にするために，階層をどのように再構成したらよいだろうか。

演習 5-8[†]　演習 4-10 のポインタリストの総称型の実装を修正して，新しいリスト型を生成する際に，生成される追加コードを減らせ。

演習 5-9　あるクラス型へ定数の集合をひとまとめにして供給するには，継承をどのように使ったらよいかを説明せよ。

演習 5-10　a) タイピングスピード，b) システムにログインしている利用者の数，c)[†] 構文木に割り当てられている節数，d) 二項演算子節の単項演算子節に対する相対的な割合，これらを監視するためのクラス型をそれぞれMonitor 型を使って作成せよ。

演習 5-11　Monitor 型を多重継承を使わないように書き直せ。両者を比べたときの各実装を長所は何か。2 つの実装は等価だろうか，動作の違いはあるだろうか。

演習 5-12　以下のプロトコルクラスを考えよ。

```
class OnOff {
  protected:
   OnOff();
  public:
   virtual ~OnOff() {}
   virtual void turnOn() = 0;
   virtual void turnOff() = 0;
};
```

多重継承を使って，演習 5-11 のMonitor 型の単一継承版にOnOff プロトコルを追加せよ。新しいMonitor 型を派生させ，以下のドライバで，プロトコル応答を検査せよ。

```
void
controller( OnOff &thing ) {
   while( 1 ) {
      char buf[ 81 ];
      cin >> buf;
      if( strcmp( buf, "off" ) == 0 )
         thing.turnOff();
      else if( strcmp( buf, "on" ) == 0 )
         thing.turnOn();
   }
}
```

演習 5-13 継承を使って，ある基底クラスの異なる「外面」を提供するために，どのような手法があるかをまとめ，それらを説明せよ。従業員データベースのデータベースレコードをデザインし，a) 通常の用途に，b) 従業員の給与台帳に，c) 従業員の管理に使うようなレコードとして，また，d) FBI(連邦捜査局)が使うようなレコードとして適切な派生クラス型を用意せよ。

演習 5-14 演習 5-13 の継承階層と同じ目的の従業員データベースレコードへアクセスするための制御抽象を書け。各アプローチの長所と短所は何だろうか。

演習 5-15 sizeof を使って，以下の各クラスのオブジェクトが占めるバイト数を調べよ。

```
class C1 { int i; };
class C2 { int i; void f(); };
class C3 { int i; virtual void f(); };
class C4 { int i; virtual void f1(); virtual void f2(); };
```

この結果から，各 C++ の処理系による仮想関数と非仮想関数の実装に関して，どういった仮定をすることができるだろうか。

演習 5-16 sizeof を使って，以下の各クラスのオブジェクトが占めるバイト数を調べよ。

```
class V1 { int i; };
class V2 { int i; };
class C1 : virtual public V1 { int i; };
class C2 : virtual public V1, virtual public V2 { int i; };
```

この結果から，C++ の処理系による仮想基底クラスの実装に関して，どういった仮定をすることができるだろうか。

演習 5-17 適切に売り買い処理を発行しつつ，株価の監視を行う型をデザインせよ。

演習 5-18† 仮想コンストラクタを宣言することが正しいとすると，これは何を意味するだろうか。仮想コンストラクタの概念を実装するテクニックを考案せよ。

演習 5-19 この章のNode 階層でのdup メンバ関数を実装する原理として，演習 5-18 の解法を使ってみよう。ただし，dup メンバは，抽象構文木の所定の節で呼び出されると，指定された節を根とする部分木の複製へのポインタを返すものとする。

演習 5-20　永続的オブジェクトとは，ほかのアドレス空間に移動させることができるオブジェクトのことである。たとえば，2次記憶に書き出され，後でほかのメモリ位置に読み戻されることができたり，ネットワーク経由で伝送されることができたりするオブジェクトは永続的オブジェクトと呼ばれる。クラスオブジェクトのための汎用の永続機構を実装する際には，継承機構によって，いくつかの難しい問題が持ち上がる。これらの問題の中には，基底クラスのポインタによって参照される派生クラスを復帰させる必要性や，クラスオブジェクトで仮想関数と仮想基底クラスを実装している内部機構を復帰させる必要性といったものがある。

汎用の永続機構のデザインの概略を示せ。このアプローチで，仮想コンストラクタ，仮想基底クラス，コピーコンストラクタを使うことを考えよ。

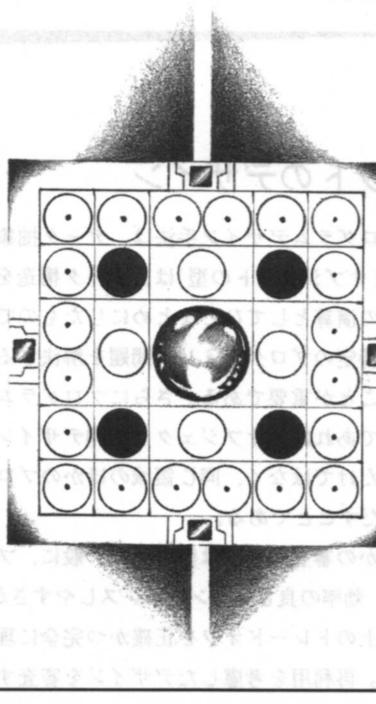
演習 5-21　この章の電卓の抽象構文木に,「制御の流れ」構造を追加せよ。抽象構文木を作成し，修正することを許し，さらに結果の構文木を実行する手段を与えるクラスをデザインせよ。この抽象構文木に適合する具象構文を持つ電卓言語をデザインし，具象構文を表現する文字列から抽象構文木を生成するコンストラクタを書け。

各自の経験を参考にして，C++のプログラムから呼び出すことができる Lisp と Smalltalk のインタプリタを覆い隠した Lisp クラスと Smalltalk クラスのデザインの概略を示せ。さらに，各自の経験を参考にして，C++のプログラムから呼び出すことができる C++ インタプリタのデザインの概略を示せ。

演習 5-22†　この章の`Gtab`の実装で，定数の代わりにクラススコープの列挙子`TSZ`が使われた理由を述べよ。

6

オブジェクト指向プログラミング



手続き型プログラミングテクニックでは，問題を解決するためのアルゴリズムにのみ注目し，プログラムの分割された構成要素としての関数の演算対象であるデータ構造は取り残されていた。これとは対照的に，オブジェクト指向プログラミングでは，プログラムの作成対象である問題領域全体に注目し，プログラムデザインの要素を問題表現のオブジェクトに対応させる。オブジェクト指向プログラミングのアプローチとしては，オブジェクトの型の集まりを定義することが一般的である。オブジェクトの型とは，問題の要素を表現するデータ構造と，解答を得るために必要とされる演算を統合したモジュールである。プログラムでは，これらのオブジェクトの型を定義し，特定の問題に対する実際のオブジェクト(インスタンス)を生成して，処理を行うための各演算を呼び出す。

C++では，クラスがオブジェクトの型としての役割を果たし，メンバ関数は演算を組み込んでいく手段を与える。

6.1　オブジェクトのデザイン

オブジェクト指向のプログラムデザイン手法は，データ抽象の使用法を拡張したものである。抽象データ型(オブジェクトの型)は，データ構造を隠すだけでなく，すべての処理をオブジェクトの演算としてひとまとめにしたものである。オブジェクト指向のデザイン手法では，特定のプログラミング問題を解決するのに，最も適したオブジェクト型を見つけ出すことが重要である。さらにプログラムが同じ分野のいくつかの問題を代表しているのであれば，オブジェクト指向デザインの目標は，当面の問題を解決するのに使われるだけではなく，同じ領域のほかのプログラムでも再利用できるオブジェクトを見つけだすことである。

デザインが適当かどうかの審査と比較は難しい。一般に，プログラムデザインの目標には，正しさのほかに，効率の良さとメンテナンスしやすさがある。通常，プログラムの必要条件とデザイン上のトレードオフを正確かつ完全に理解するにはいくらかの実験を試みる必要がある。再利用を考慮したデザインを審査することは，特定の問題

の特徴の一般性を取り出し，さらに将来のプログラムでの必要性を予想する必要もあるので，非常に難しい。また，再利用を考慮したデザインでは，もとのデザイナではないプログラマが容易に理解できるものでなければならない。

いくつかのデザイン法則には，手本にすべき経験則が存在することがわかっている。ただし，プログラムの規模が大きい場合は，問題の特徴が入り組んでいるので，実質的には，独自にデザインする必要があることが多い。このため，汎用のデザイン法則を適用する場合には注意が必要である。また，デザイン上の品質のうち，プログラムの効率性，メンテナンスしやすさ，構成要素の再利用性といった事実上の結果に関係するものは，優雅さ，簡潔さ，一貫性といった美的感覚のある言葉で表現される。すぐれたソフトウェアデザインは常識と美的感覚に依存するのであるが，問題と解法の組織立てを行うときに役に立つツールとテクニックもいくつか存在する。

オブジェクト型のデザインに関するガイドラインをいくつか論じる前に，整数のリストをソートするためのオブジェクト指向のプログラム例を２つ見てみよう。デザインに注目するために，実装の細部は示さない。第１のプログラムでは，リストをソートする関数と，出力する関数を持つ抽象データ型Intlistを使っている。整数は，Intlistオブジェクトの生成時にリストに読み込まれる。

```
class Intlist {
    // etc.
  public:
    Intlist();
    sort();
    write();
};

main() {
    Intlist *ilist = new Intlist();
    ilist->sort();
    ilist->write();
}
```

第２のプログラムでは，抽象データ型Sortedintlistを使っている。整数は，リストオブジェクトの生成時に読み込まれ，同時にソートされる。Sortedintlistは，出力のための関数だけを持つ。

```
class Sortedintlist {
    // etc.
  public:
    Sortedintlist();
    write();
};

main() {
    Sortedintlist *slist = new Sortedintlist();
    slist->write();
}
```

デザインとしてはどちらが優れているだろうか。あるいは，両者ともに同じぐらい優れているだろうか。

オブジェクト型のデザインで非常に役に立つガイドラインは，解決しようとする問題を表現したものから，名詞と動詞を探すことである。名詞はオブジェクト自体を示し，動詞は演算の振舞いを示す。この法則を「整数のリストをソートする」という問題に適用すると,「ソートする」という演算を持った「整数のリスト」オブジェクト型を使うようにプログラムをデザインすべきだということがわかる。第1のIntlist型は問題表現が暗示する「生来の」解法に比較的に近いが，第2のSortedlist型はこの解法から直接考え出されたものではない。このSortedlist型が，リストの作成時に行うソートのオーバヘッドは，ただのリストを汎用の記憶機構としてだけ使いたいプログラムには適していない。

ソート済みリスト型(Sortedlist)は，ほかの問題，特に汎用のリスト構造の状況では，非常に適した型であるかもしれない。たとえば，多種類のリスト処理を実行するプログラムでは，ソート済みリスト型のほうがリストの探索や併合を非常にうまく行うことができることもあるだろう。また，重複する要素を持たないリストを表現する圧縮済みリスト型(Compressedlist)というものも考えられるが，これもまたある状況では優れたリスト処理を与えることができるだろう。

このことは，オブジェクト指向デザインのもう1つの特徴，つまりオブジェクトの基本型から専用版を作成するための継承機構を導き出す。オブジェクト指向デザインでは，各アプリケーションごとにまったく新しいオブジェクト型を作成するのではなく，基本型から新しい型を構築していく。このリスト処理プログラムでは，基底クラスとしてListを使い，Listから派生したクラスとしてSortedlistあるいはCompressedlist

を持てばよいだろう。基底クラスでは，演算sortを仮想関数として宣言することもできる。この場合，Sortedlistのsortは(すでにソートされているので)何も実行する必要がないし，Compressedlistには(基底クラスのものが使えばよいので)sortを新たに用意する必要すらない。

基底クラスと専用版としての派生クラスの間の継承関係は，オブジェクト指向デザインの専門用語では，is-a(…は…でもある)関係として知られている。したがって，SortedlistはListでもあり，同じ演算を持ち，Listが使われるところであればどこで使われてもよい。継承機能を使うことによって，専用版としての派生クラスの動作の中から一般的な特性が分離され，基底クラスにまとめられる。このため，基底クラス型に対して書かれたコードは，派生クラスオブジェクトすべてに対して使うこともできる。

オブジェクト指向デザインにおいて，どの部分で継承を使って型の専用版を作成すればよいかというガイドラインは，プログラミングの問題領域に関する表現から形容詞を探すことで得られる。この例では「ソート済みのリスト」と「圧縮済みのリスト」が派生クラスを使うところを指示している。

オブジェクト指向デザインでは，総称型の存在を探すことも重要である。再度「整数のリスト」,「文字列のリスト」,「従業員レコードのリスト」というリスト処理プログラムを考えよう。これらすべてのリストは，要素の型が違うけれども単一の総称リスト型から生成し，構築することができる。総称型の存在を見つけ出すための手がかりは，総称型がほかの型を格納するための箱(container)であり，格納物を構成するという機能だけを持つということであろう。しかしプログラムが，総称型の要素として複数の型を使ったり，あるいは総称型をほかのプログラムで再利用したりする場合を除くとすれば，総称型を構築する意味はほとんどないだろう。

6.2 オブジェクトの発見

問題表現から，プログラムに実装するために最も適したオブジェクト型を常に見つけ出すことができるとは限らない。多数のオブジェクトが1つの基本型の実装を共有できるように，これらに共通する1つの抽象表現を見つけ出すことは，かならずしも簡単な仕事ではない。プログラムのデザインは，問題の正しい表現，抽象概念の抽出，

プロトタイプの実装といった何らかの試みが必要とする試行錯誤の結果として完成することが多い。しかし，これは特にオブジェクト指向デザインに限ったことでも，新しいことでもない。

オブジェクト指向プログラミングの新しいことは，アルゴリズムの実装としてではなく，オブジェクト型の実装としてプログラムを考える手法である。あるプログラマにとっては，このように発想を変えることは難しいだろうし，また，特に架空の抽象概念をオブジェクト型の候補として考えることも難しいだろう。

私達が慣れ親しんでいる実世界の実在物に相当する物理的なオブジェクト型を想像することは容易である。たとえば，ディスクドライブあるいはテープドライブといったデバイスは物理的なオブジェクトである。デバイスはオペレーティングシステムのカーネル領域におけるオブジェクトの1つである。第5章では，デバイスドライバへの汎用的なインターフェイスの役割を果たすクラスの例を示した。

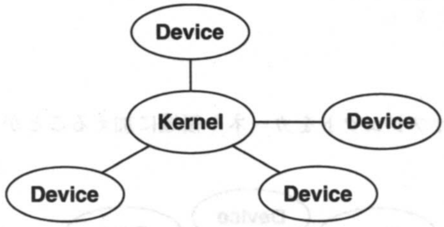
```
class Device {
  protected:
   Device();
  public:
   virtual ~Device();
   virtual int open( char *, int, int );
   virtual int close( int );
   virtual int read( int, char *, unsigned ) = 0;
   virtual int write( int, char *, unsigned ) = 0;
   virtual int ioctl( int, int ... ) = 0;
};
```

このクラスは，オブジェクト指向デザインによるオペレーティングシステムでは，すべての種類のデバイスの基本オブジェクト型と考えられる。この基本型から継承を使って，ディスクデバイスやテープデバイスを表現する多種類の型を作成することができる。

```
class Disk : public Device {
  public:
   int open( char *, int, int );
   int close( int );
   int read( int, char *, unsigned );
   int write( int, char *, unsigned );
   int ioctl( int, int ... );
   Disk();
   ~Disk();
};
```

```
class Tape : public Device {
  public:
   // etc.
};
```

ここに多種類のデバイス型があるが，カーネルからはこれらのデバイスオブジェクトがすべて同じものに見える。



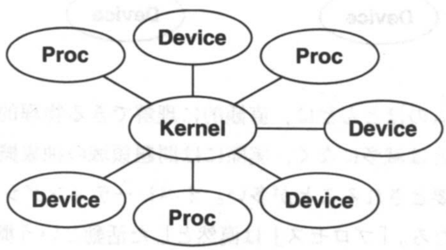
ただし，プログラムのほとんどは，直観的に理解できる物理的なオブジェクトの表現を直接取り扱うことは滅多になく，実際には問題領域の抽象概念の一部を表現するオブジェクト型が必要とされることが多い。オペレーティングシステムの例では,「プロセス」がこれにあたる。「プロセス」は漠然とした活動という概念であるが，これもオブジェクトとして表現される。

いわゆるオペレーティングシステムのカーネルは，ばらまかれたデータ構造と多数のデータ構造を演算するための関数によってプロセスを表現している。オブジェクト指向デザインでは，プロセスに相当するオブジェクト型を作成するために，これらのデータ構造と関数は一緒にクラスにまとめられる。このオブジェクト型へのインターフェイスは，カーネルがプロセスを制御するための関数の集合である。以下に示すクラスProcは，プロセスをある優先権でイベント待ち状態で眠らせる関数(sleep)，眠っているプロセスを起こす関数(wakeup)，動作中のプロセスの状態を保存したり(save_runstate)，もとに戻したり(resume_runstate)する関数，そしてプロセスへメッセージ番号フォーマットでメッセージを送る関数(send_msg)を持つ。プロセスを分岐複製させるために，既存のProcから新しいProcを作成するためのコピーコンストラクタも用意されている。デストラクタは，終了したプロセスをシステムから取り除く。システムの実記憶と仮想記憶のプロセスを交換させるための演算は，Proc

オブジェクトの隠された機能である記憶管理機構の一部と考えられるため，このインターフェイスでは提供されていない。

```
class Proc {
  public:
    int sleep( Sleepq *, int );
    void wakeup();
    int save_runstate();
    void resume_runstate();
    int send_msg( int );
    Proc( Proc & );
    ~Proc();
};
```

これで，プロセスオブジェクトをカーネル領域に加えることができる。



さらにカーネルのことも考えてシステムデザインを続ける場合，次に，カーネルとデバイスおよびプロセスオブジェクト間の関係の特徴を決定する必要がある。デバイスとプロセスはカーネルの構成要素だろうか，あるいは，カーネルの演算対象で，独立した実体だろうか。これらの関係は，オブジェクト指向デザインの専門用語では，has-a関係とuses-a関係として知られている。カーネルオブジェクトの一部にデバイスとプロセスを組み入れるように決定するのであれば，カーネルを実装するクラスは，これらの部分を表現するためのデータメンバを持つだろうし，カーネルがデバイスオブジェクトとプロセスオブジェクトを演算対象として使うだけという決定を下すのであれば，カーネルはオブジェクト指向のデザインであっても，手続き型のデザインであってもよいだろう。C++では，両者を混成したプログラムデザインも許され，オブジェクトはほかのオブジェクトからだけではなく，非メンバ関数からも操作されることができる。オブジェクトは，わかりにくいプログラムデザインをわかりやすくするための抽象表現

である。これに対し，手続きの抽象表現は，オペレーティングシステムの中央制御機能のデザインといった，ある種のプログラミング問題を解く際にもうまく使われている。

カーネルデザインの問題を扱う際には，実際には，さらに解析を行う必要がある。ここでは，デバイスとプロセスとの関係の解説をわかりやすくするために，カーネルが単一要素であるという仮定を行っていた。このため，カーネルに望ましい機能を調査し，複数のオブジェクトを見つけ出すか，あるいは適切だと思われるのであれば手続き型テクニックを使ってさらに分解する必要がある。カーネルデザインに追加すべきオブジェクトの候補としては，状態，割り込み，バッファ，ファイルに相当するものがある。また，デバイスとプロセスのオブジェクトの初期デザインも，新たに精製したカーネルとの関係に対応するように修正が必要だと思われる。この際に，いくつかの取捨選択を行う必要があるだろうし，システムデザインを完成に向けて発展させるために初期の考えを変更させる必要もあるだろう。

機能分解によるデザインに慣れているプログラマは，オブジェクト型を使ったデザインに慣れるためにいくらか実地経験が必要である。しかし，いったんオブジェクト指向デザインの意味を理解したら，次は使いすぎに注意すべきである。リストをソートする代わりにソート済みリストを作成した先の例のように，あらゆる目的に満足するだけオブジェクト型を考え出すことが可能である。しかし，無計画にオブジェクト型の種類を急増させてしまうと，プログラムデザインが不明瞭になる。よいオブジェクト指向デザインとは，プログラムが解決しようとする問題に明確に対応し，かつわかりやすく，実装しやすく，さらにメンテナンスしやすいもののことである。

6.3 モジュールとしてのオブジェクト型

オブジェクト指向プログラミングでは，問題の要素に対応するデザイン要素であるオブジェクト型によってプログラムモジュールが構築される。このプログラム構築法の長所は，プログラム構築のすべての段階を通じて概念的に連続性があり，結果として生じるオブジェクト型モジュールを容易に再利用できることである。

プログラムの概念的な構造は，問題表現と実装の両者で同一のものが保持されるだけでなく，デザインの精製段階でも同じものが保たれる。最初にオブジェクト型が見つけ出され，これらの型に細部が追加されることで，デザインが精製される。これは，高

レベルのデザインがほかの構造へ分解されるという意味において，関数のトップダウン式の段階的精製とは対照的である。細部をすべて知る前にデザインの構造が完全にわかれば，プロトタイピングを素早く行うことができる。オブジェクト型のプロトタイプの実装は，ある程度の水準にある演算を最低限の個数用意し，未完成の細部は隠すことで行われ，検査が必要とされるデザインを早めに試してみることができる。プロトタイプを早めに作成できることは，デザインの試行錯誤のコストを抑えることにもなる。

ここで，文字列処理プログラムを考えよう。プログラムデザインの際に文字列オブジェクト型が必要であることがわかると，第3章の最初のクラスStringの例のように，やさしいデータ構造と既存のライブラリ関数を使ってクラスStringをすばやく実装できる。このStringの簡易版は，どういった文字列処理が必要かを正確に調べるための布石として使えるだろう。プロトタイプのクラスStringに処理を追加することは，プログラムデザインの細部を精製することに相当する。よりよいデザインの可能性を確認できたら，次は第4章で行ったように，高機能版のStringの実装でプロトタイプ版を置き換えればよい。

オブジェクト指向デザインはトップダウン式ではないので，結果として得られるプログラムモジュールは比較的独立しており，再利用が容易である。このモジュールは特定のプログラムのデザイン階層とは無関係であるから，特定のプログラム構造には依存しない。オブジェクト型によるモジュールは，問題解決のための補助機能の代わりに，特定の問題領域の要素を実装したものである。したがって，同一の問題領域を対象とするプログラムはもちろん，問題領域の要素が重複するプログラムであれば，このモジュールをあらゆるプログラムから使うことができる。たとえば，Stringは単語や文章を表現する型を必要とするすべてのプログラムから再利用できる。

モジュールの再利用のもう1つの特徴として，継承を使うことでオブジェクト型を容易に拡張したり，専用版を作成したりできることがある。新しい機能を追加して派生クラス型を作成する際に，もとのモジュールに手を入れることはない。汎用のオブジェクト型としてのモジュールは，多種の問題で再利用することができ，基底クラスの一般的な特性だけを使ったコードは，新しい専用版に対しても動作する。

6.4　動的なオブジェクト指向のスタイル

オブジェクト指向プログラミングスタイルは，オブジェクトの実行時生成と，オブジェクトに対する演算の動的束縛(呼び出し手続きの動的決定)から構成されるプログラミングスタイルと関連が強い。このプログラミングスタイルは，Flavors あるいは Smalltalk といった対話型オブジェクト指向言語とシステムに端を発するものである。これらのシステムは，実行時の柔軟性を重視してデザインされ，プロトタイピング，問題の即時解決に使われるツールとして有効であることが立証されている。

C++を使って，すべてのオブジェクトを実行時に生成し，オブジェクトに束縛する演算をすべて仮想関数で動的に対応させるオブジェクト指向プログラムを書くことも可能である。オブジェクトの生成と演算の実行は，コンパイルされたプログラムのコードに従って行われる。仮想関数の呼び出しは動的に束縛されけれども，型検査は静的に行われる。このため，C++は対話型の柔軟性には欠けるが，コンパイル時の強い型検査と実行効率にによって，この短所を埋めている。

動的なオブジェクト指向のプログラミングは，シミュレーションに使われることが多い。オブジェクト型は，問題領域の実在物をシミュレートするように実装される。次に生成されたオブジェクトが動き出してシミュレーションを演じる。各オブジェクトは，ほかのオブジェクトの演算を呼び出したり，シミュレーションに参加する新しいオブジェクトを作成したりして，相互作用を行う。本章の以下の部分では，空港での飛行機発着をシミュレートするプログラムを示す。

この空港のシミュレーションでは，C++の「タスク(task)」ライブラリを使って，複数の動作をコルーチンとして実現している。このライブラリは，シミュレーションのための時間を管理し，複数のルーチンの実行割り当てを制御する機構を与える。このシミュレーションでのコルーチンは，クラスtaskと，taskから派生したクラスのコンストラクタである。taskオブジェクトが作成されると，コンストラクタの実行がライブラリの制御下に組み込まれる。タスクライブラリは，均等に時間を割り付けるように実装されているので，ある時間が経つと，各タスクは自分の実行を止め，ほかのタスクに実行の機会を与える。このシミュレーションでのタスクの実行は，taskのメンバ関数

```
void task::delay( int )
```

に制御される。この関数は，所定の時間単位が経つと実行を中断し，ライブラリに次に実行するコルーチンを割り当てさせる。経過時間はライブラリ中の大域変数clockに記録される。関数

```
void task::cancel( int )
```

はタスクを終了させる。

このライブラリは，待ち行列を構築するためのクラスも提供している。複数のタスクに対して，待ち行列にオブジェクトを追加したり，逆に取り出したりという機能を独立に働かせる必要があるので，タスクライブラリにクラスqheadとqtailが分かれて存在している。待ち行列に関する演算は次の通りである。

```
qtail *qhead::tail()
```

は，各qheadに結び付いているqtailを得るための関数であり，

```
int qtail::put( object * )
```

は，待ち行列の最後にオブジェクトを追加する関数であり，

```
object *qhead::get()
```

は，待ち関数の先頭からオブジェクトを取り出す関数である。待ち行列がいっぱいであるか空であるかを調べるには以下の関数を使う。まず

```
int qtail::rdspace()
```

は，待ち行列にオブジェクトを追加するための空き場所がどのくらい残っているかを返し，

```
int qhead::rdcount()
```

は，待ち行列中のオブジェクト数を返す。待ち行列の要素はすべて基底クラスobjectから派生したクラスである。タスクライブラリのtaskを始めとするクラス群は，階層の根に基底クラスobjectを持つため，これらの型のオブジェクトはすべて待ち行列に置くことができる。

タスクライブラリは乱数を生成するクラスも用意している。次の関数

```
int urand::draw()
```

を使うことで，所定の範囲内で一様に分布した乱数を提供するクラスurandを使うことができる。

関数mainは，Airportタスクオブジェクトを生成し，このオブジェクトを一定時間実行させ，シミュレーションを停止させた後，復帰する。関数mainは，Airportを生成するとすぐに，自分もコルーチンとしてほかのタスクとともに走るタスクに変わる。タスクシステムポインタthistaskは，現在実行中のタスクを常に参照するもので，関数mainの実行をコルーチンとして制御するために必要である。

```
#include "Airport.h"

main() {
   Airport *ap = new Airport;
   thistask->delay( 5000 );
   delete ap;
   thistask->resultis(0);
}
```

Airport.hは，タスクライブラリのヘッダファイルをインクルードしている。シミュレーションのためのオブジェクト型は，すべてタスクライブラリの基底クラスobjectから作成されている。

```
/* Airport.h */
#include "task.h"

class Plane : object {
   static int fltcount;
   long start;
   int fltno;
  public:
   Plane() { fltno = ++fltcount; }
   long howlong() { return clock - start; }
   void set() { start = clock; }
   int flt() { return fltno; }
};
```

```
class PlaneQ {
    qhead *head;
    qtail *tail;
  public:
    PlaneQ() {  head = new qhead( ZMODE, 50 );
             tail = head->tail();}
    void put( Plane *p ) { p->set();
             tail->put( (object*)p );}
    Plane *get() { return (Plane *)head->get(); }
    int isroom() { return tail->rdspace(); }
    int notempty() { return head->rdcount(); }
};

class AirControl : public task   {
    PlaneQ   *landing, *inair;
  public:
    AirControl( PlaneQ *, PlaneQ * );
};

class GroundControl  : public task {
    PlaneQ *takingoff, *onground;
  public:
    GroundControl( PlaneQ *, PlaneQ * );
};

class Airport  : public task {
    PlaneQ   *takingoff, *landing, *inair, *onground;
    AirControl *acontrol;
    GroundControl *gcontrol;
  public:
    Airport();
    ~Airport();
};
```

オブジェクトの型には，飛行機のためのPlane，空港の施設を使うために待機中の飛行機の待ち行列であるPlaneQ，空港の上空の飛行機を指揮するAircontrol，地上の飛行機を指揮するGroundcontrol，これらの飛行機の待ち行列と空中と地上の交通の指揮系統から構成されるAirportがある。Airport自体は滑走路を制御する。

Planeは，タスクシステムで扱われる基底クラスobjectからの派生クラスであり，タスクライブラリ変数clockを参照してシミュレート時間を測定し，飛行時間と遅れ情報を記憶する。

PlaneQは，飛行機を待機させるための待ち行列であり，qheadとqtailをまとめたクラスである。PlaneQは，飛行機を50機まで待機させることができ，空の待ち行列へgetを実行した場合はヌルポインタを返すように設定されている(空の待ち行列へgetを実行した場合に，呼び出したタスクが一時停止するように実装することも可能である)。

AirControl，GroundControl，Airportは，すべてtaskである。これらの型のオブジェクトが生成されると，これらのコンストラクタはコルーチンとして動作する。

Airportのコンストラクタは，サービスを待つ飛行機の待ち行列を作成し，待ち行列を管理する制御タスクを作成し，最後に，Airportタスクが取り消されるまで走り続ける無限の繰り返しに入る。空港は10時間単位ごとに1機の着陸と1機の離陸を行う。ただし，待機中の飛行機が不在であれば，待ち行列から飛行機を取り出す。飛行機の処理状況の報告は，printf関数を使って行う。着陸を待っている間に飛行機が燃料切れを起こした場合は，すこし荒っぽいが，飛行機は墜落したものと考えて，以降のシミュレーションから取り除くことにする。

```
Airport::Airport() {

    takingoff = new PlaneQ;
    inair = new PlaneQ;
    landing = new PlaneQ;
    onground = new PlaneQ;

    acontrol = new AirControl( landing, inair );
    gcontrol = new GroundControl( takingoff,onground);

    Plane *tp=0, *lp=0;  // 待機中の飛行機
    int maxwait = 30;

    for(;;){
       delay( 10 );
       if( !lp ) lp = landing->get();
       if( !tp ) tp = takingoff->get();
       if( lp ){
          if( onground->isroom() ) {
             printf( "flight %d landing \n",
                lp->flt());
```

```
               onground->put( lp );
               lp = 0;
            }
            else if( lp->howlong() > maxwait ) {
               printf( "flight %d crash landing!\n",
                  lp->flt());
               delete lp;
               lp = 0;
            } else
               printf( "flight %d landing delayed \n",
                  lp->flt());
         }
         if( tp ) {
            if( inair->isroom() ) {
               printf( "flight %d taking off\n",
                     tp->flt());
               inair->put( tp );
               tp = 0;
            } else
               printf( "flight %d take off delayed \n",
                     lp ->flt());
         }
      }
   }
```

Airportデストラクタは，Aircontrolタスクを取り消し，以後，飛行機の着陸を新たに予定に組み込まないようにする。さらに，待機中のすべての飛行機が着陸してしまうまで待ってから，残りのタスクを取り消す。

```
Airport::~Airport() {
   acontrol->cancel(0);
   while( landing->notempty() )
      thistask->delay( 10 );
   gcontrol->cancel(0);
   printf("Airport Closed\n");
   cancel(0);
}
```

Aircontrolは，1～30単位時間ごとに，到着した飛行機が着陸できるように処理を行う。到着時間の間隔が変わるようにするために，タスクライブラリの乱数生成クラスurandを使う。この例では，ただ1つの空港だけをシミュレートしているため，着陸の待ち行列に置かれる飛行機は，空港から離陸してしまった飛行機が再利用されるか，あるいは新規に作成される。

```
AirControl:: AirControl( PlaneQ *landing,
            PlaneQ *inair ) {

   urand n( 1, 30 );

   for(;;){
      delay( n.draw() );

      Plane *p = inair->get();
      if( !p ) {
         p = new Plane;
      }
      if( landing->isroom() ) {
         printf ("flight %d arriving for landing\n",
            p->flt() );
         landing->put( p );
      } else {
         printf ("flight %d redirected\n",
            p->flt() );
         delete p;
      }
   }
}
```

Groundcontrol は，地上の飛行機に対して 1～30 単位時間ごとに離陸の処理を行う。

```
GroundControl::GroundControl( PlaneQ *takingoff,
                PlaneQ *onground ) {

   urand n( 10, 30 );
   Plane *p = 0;

   for(;;){
      delay( n.draw() );
      if( !p ) p = onground->get();
      if( p ) {
         if( takingoff->isroom() ) {
            printf ("flight %d leaving gate\n",
                  p->flt() );
            takingoff->put( p );
            p = 0;
         } else
            printf ("flight %d delayed at gate\n",
                  p->flt() );
      }
   }
}
```

以下に，空港シミュレーションの出力の最後の部分を示す。

```
flight 5 arriving for landing
flight 82 crash landing!
flight 84 arriving for landing
flight 9 landing delayed
flight 48 leaving gate
flight 9 landing
flight 48 taking off
flight 48 arriving for landing
flight 17 leaving gate
flight 83 landing
flight 17 taking off
flight 17 arriving for landing
flight 5 crash landing!
flight 84 crash landing!
flight 32 leaving gate
flight 48 landing
flight 32 taking off
flight 17 crash landing!
flight 60 leaving gate
Airport Closed
```

シミュレーションの結果，墜落が頻繁に起きている。これは着陸待ちが多すぎたからではなく，遅れの許容時間が短すぎたことや，離陸の待ち行列が大きすぎたことが原因だと思われる。最大着陸遅延時間，地上サービス待ち時間，到着間隔，待ち行列の大きさにほかの値を試みることで，墜落，遅れ，到着先変更の発生にどういった影響があるか知ることができる。

演　習

演習6-1　空港シミュレーションに対して，緊急着陸をうまく扱う手段を追加せよ。

演習6-2　本章の最初の節の例で使われた抽象データ型`Intlist`を実装せよ。`Intlist`から派生した型として`Sortedintlist`を実装せよ。

演習6-3　制御型`control`の専用版として，空港シミュレーションの空中と地上の制御モジュールを再びデザインせよ。

演習 6-4† この空港シミュレーションに，複数の空港間の飛行経路を決定する国内飛行管制も組み込み，拡張せよ。

演習 6-5 アプリケーションは，初期状態から始まって，入力に応じた一連の状態間を移動するという考えを使って作られていることが多い。各状態では，何らかの動作が行われる。たとえば，自動預金プログラム，航空予約システム，コンパイラ語彙解析プログラム(レキシカルアナライザ)は，この考えで概念をまとめることができる。一般的に，状態の遷移は，現在の状態を分類する表として表現することができる。入力には，対応する整数のコードが割り当てられる。各状態には，新しい状態のための動作が対応付けられている。

このアプローチの長所と潜在的に存在する問題は何か。

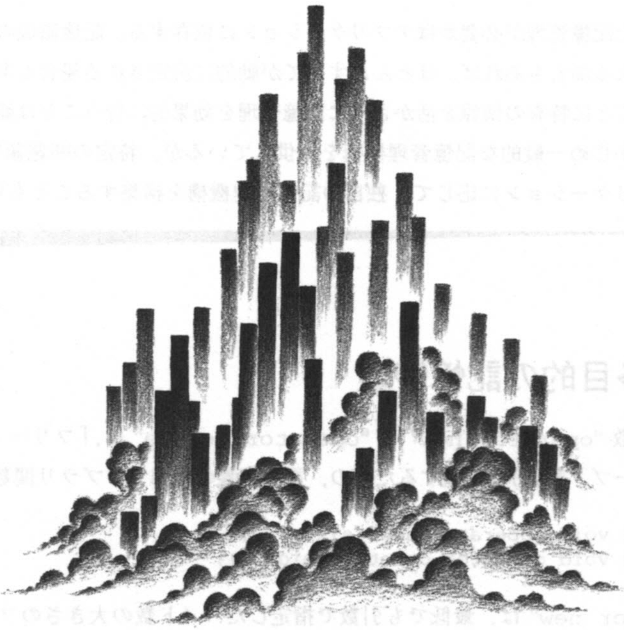
この手法に代わる枠組みとして，所定の状態のすべての意味(後続する可能性な状態と動作も)をクラスにまとめた状態オブジェクトを作成できる。

この場合のオブジェクト指向アプローチの長所と短所は何か。

自動預金アプリケーションをデザインし，これら2つのアプローチを使って実装せよ。

7

記憶管理



記憶管理は，プログラミングに関して論じた教科書の多くでは，重要な話題とはされず，言語やプログラミング環境が扱う領域と考えられることが多い。しかし優れたアプリケーションを作成するためには，正しい記憶管理の制御は不可欠であり，その効率は重要である。

メモリの割り当てや解放の効率が悪いと，メモリを得ようとする冗長なシステムコールが多数発生したり，利用できる記憶領域が断片状に分解されて実メモリと仮想メモリの間でメモリページが頻煩に交換される「スラッシング」(メモリページの交換作業だけに時間を取られてしまい，プログラムを実行できなくなる)と呼ばれる状態を引き起こしたり，メモリを使い切ってしまったりして，アプリケーションの効率をひどく悪くする。さらに，こうした効率の悪さに対処しようとした結果，記憶領域の解放が早すぎたり，適当に別名を付けてしまうといった，間違った記憶管理がもたらされることもある。

どういった記憶管理が必要かはアプリケーションに依存する。記憶領域のすべてが静的に決定される場合もあれば，ほとんどすべてが動的に決定される場合もある。アプリケーションごとに特有の情報を活かさずに記憶管理を効果的に行うことは難しい。C++では，あらかじめ一般的な記憶管理機構を提供しているが，特定の問題領域，クラス，またはアプリケーションに応じて，独自の記憶管理機構を構築することも可能である。

7.1　多目的の記憶管理

演算子関数 “operator new” と “operator delete” は,「フリーストア」(いわゆる「ヒープ」領域)を管理するための，言語定義の標準ライブラリ関数である。

```
extern void *operator new( size_t );
extern void operator delete( void * );
```

“operator new” は，最低でも引数で指定したバイト数の大きさのフリーストア領域を返し，“operator delete” は “operator new” が割り当てたフリーストア領域を解放する。

```
void
allocer() {
   Thing *tp = new Thing;
   dostuffwith( tp );
   delete tp;
}
```

ここで確保されるオブジェクトの大きさはnew演算子のオペランドから決められ，“operator new”に暗黙の引数として渡される。ほかの演算子関数と同じように，以下のように“operator new”と“operator delete”を明示的に実行することも可能ではあるが，

```
int *
hunkoints()
   { return (int *)operator new(sizeof(int)*16384); }
```

以下の前置形式のほうが，一般的にわかりやすく，安全である。

```
int *
hunkoints()
   { return new int[16384]; }
```

記憶管理を細かく制御するために，“operator new”の処理が失敗した場合に呼び出される関数を，標準関数set_new_handlerを使って設定することができる。set_new_handlerの引数は，引数を持たず結果を返さない関数へのポインタである。set_new_handlerは前回の引数を戻り値として返す。要求されたメモリを確保できるか，あるいはset_new_handlerにヌル引数が与えられるまで，“operator new”はset_new_handlerで設定された関数を呼び出し続ける。

```
void
give_up() {
   error( "ran out of memory!" );
   exit( 1 );
}

void
scavenge() {
   garbage_collect();
   set_new_handler( give_up );
}
```

```
void
allocer() {
    void (*saved_handler)() = set_new_handler(scavenge);
    Thing *tp = new Thing;
    dostuffwith( tp );
    set_new_handler( saved_handler );
    delete tp;
}
```

この場合，“operator new”は要求されたメモリの確保に失敗した場合，メモリを探し出すために最初にscavengeを呼び出す。scavengeが要求を満たすだけのメモリを探し出すことに失敗すれば，give_upがメッセージを表示してプログラムを終了させる。

ある環境では，デフォルトのnew_handler関数(set_new_handlerによって設定されるエラーハンドラ)の動作として，例外を送出するように設定されていることがあるが，new_handlerにヌルを設定すれば，メモリ確保の失敗時に単にヌルを返すようになる。ただし，最近の流儀としては，“operator new”が返す結果が非ヌルかどうかをプログラムコードが検査する代わりに，ほかの手段でメモリ確保エラーを処理するのが一般的である(詳しくは第8章を参照されたい)。

new_handlerは非常に柔軟な機構であり，ほとんどのアプリケーションには，この標準ライブラリの演算子newとdeleteの機能で充分である。

しかし，例外的な状況では，独自バージョンのnewとdeleteを用意して標準バージョンをオーバライドすることで，標準バージョンの代わりに，利用者定義バージョンが使われるように変更することができる。

たとえば，標準の“operator new”は，記憶領域がすべて0で初期設定されていることを保証しないが，一部のオペレーティングシステムの環境でプログラムをすこしだけ実行した場合，記憶領域がたまたま0であることもある。コードがこの表面上の0への初期設定を利用している場合，エラーはテスト時には捕らえられず，プログラムの長期運用時に現れてしまうだろう。このため，早期のテスト時にこの種のエラーを捕えるために,「ごみ」で埋めた記憶領域を返すnewとdeleteを開発バージョンとして用意したいと考えることもある。

```
extern void *malloc( unsigned );
extern void free( void * );
```

```
extern void *
operator new( size_t sz ) {
   char *p = (char *) malloc( sz );
   if( p ) {
      for( int i = 0; i < sz; i++ )
         p[i] = '~';
   }
   return p;
}

extern void
operator delete( void *p )
   { if( p ) free( p ); }
```

set_new_handlerの機能は，標準ライブラリ“operator new”の実装の一部にすぎず，このように利用者定義バージョンを定義すると(set_new_handler機構を考慮し，新しい“operator new”を実装した場合を除いて)，set_new_handlerは新しい“operator new”の動作とは無関係になることに注意してほしい。

一般的には，標準バージョンの“operator new”と“operator delete”を置換するのは止めたほうがよい。これらの演算子の標準バージョンは，多目的の記憶管理に対して非常によく出来ていて，利用者定義バージョンを使ったところで，多目的での性能を向上させることはほとんど不可能である。ある特定のアプリケーションに対しては，専用バージョンのnewとdeleteが性能を向上させる可能性もあるが，実際にはうまくいくことは稀で，ほとんどは標準バージョンの演算子newとdeleteの意味を仮定しているプログラムにバグを混入してしまうだけだろう。

ほかの多くの演算子と同じように，“operator new”を多重定義することもできる。たとえば，先に示したバージョンと同じように，確保したメモリをどういう「ごみ」で埋めるかを指定できる“operator new”を定義することもできるだろう。

```
extern void *
operator new( size_t n, const String &fill );
```

operater newのすべてのバージョンは，“void *”を返し，確保される記憶領域の最小バイト数を指定するsize_t型の第1引数を持つ必要がある。追加の引数の

型と数は自由である。どのバージョンの演算子newが呼ばれるかは，標準の多重定義決定機構によって決められる。

```
SortedCollection *cp = new SortedCollection;
delete cp;
String fill = "<pure and unadulterated garbage>";
cp = new ( fill ) SortedCollection;
delete cp;
```

この1行目では，SortedCollectionオブジェクトの記憶領域を確保するために，“operator new”の単一引数バージョンが呼び出され，そのオブジェクトの初期設定を行うためにSortedCollectionのデフォルトコンストラクタが実行される。4行目の確保では，“operator new”の2引数バージョンがマッチする。“operator new”の2番目以降の実引数は，newの直後の引数並び中で渡されるということに注意されたい。さらにオブジェクトの初期設定子があれば，型名の後に追加される。

```
String *sp = new ( fill ) String( "Hello, World!" );
```

最初の引数並びが“operator new”の引数を指定し，2番目の引数並びがStringコンストラクタの引数を指定する。

“operator new”と違って，“operator delete”を多重定義することはできない。このため，“operator delete”の単一の実装は，“operator new”のすべてのバージョンに対して正しく動作する必要がある。いい換えれば，“operator new”のすべてのバージョンは，標準バージョンの“operator delete”のために正しく動作する必要があるという意味でもある。

```
void *
operator new( size_t n, const String &pat ) {
    char *p = (char *) ::operator new( n );
    // 演習7-7参照
    const char *pattern = pat;
    if( !pattern || !pattern[0] )
        pattern = "\0";
    const char *f = pattern;
    for( int i = 0; i < n; i++ ) {
        if( !*f )
            f = pattern;
        p[i] = *f++;
    }
    return p;
}
```

この“operaor new”の多重定義バージョンの実装は，記憶管理のために，標準バージョンのnewとdeleteを使い，単に追加の動作を与えているだけである。

ありふれた例としては,「配置構文」バージョンがある。このバージョンの“operator new”は，一部のシステムでは標準として与えられることが多い。

```
void *
operator new( size_t, void *placement )
    { return placement; }
```

このバージョンの“operator new”は単に第2引数を返す。これは，特定の位置にオブジェクトを配置するのに使われる。

```
void * const ports_loc = (void *) 0x00400000;

Ports *ports = new ( ports_loc ) Ports;
```

この宣言は，位置ports_locの記憶領域をPorts型のオブジェクトとして初期設定するという効果がある。この際にも，オブジェクトの寿命の終わりでは，portsが指すPortsオブジェクトのデストラクタが実行される必要がある。通常，newで割り当てられたオブジェクトはdeleteで解放され，この際に，削除作業に先立って暗黙のうちにデストラクタが実行される。しかし，標準の“operator delete”は削除される記憶領域が標準の“operator new”で確保されているということを仮定しているので，portsが指すオブジェクトに対してdeleteを実行すると悲劇が起こる。この場合，以下のように，明示的にデストラクタを呼び出すことができる。

```
ports->~Ports();
```

配置構文バージョンの“operator new”と，明示的デストラクタ呼び出しのもう1つの使い道として，すでに割り当てられているオブジェクトを再び初期設定することがある。

```
File *f = new File( "scratchfile" );
// ... コード中，fはファイルの別名として使われる。
// ... アプリケーション中でファイルは変更される。
// 現在のファイル資源を解放する。
f->~File();
// 古いアドレスに新しいオブジェクトを作成する。
```

```
f = new ( f ) File( "new_scratch" );
// ... 実行を続ける。
delete f;    // 最後に，最初の new に対応して削除が行われる。
```

配置構文バージョンの“operator new”と明示的デストラクタ呼び出しは役に立つ機能ではあるが，エラーの原因になりやすいものでもあるので，使用にあたっては十分注意が必要である。通常，“operator new”の使用はすべて，対応する“operator delete”と組で使い，デストラクタは暗黙のうちに実行される。これに対して，配置構文を経由して「確保された」メモリを削除したり，あるいはすでに明示的に破棄されてしまったオブジェクトに対する暗黙のデストラクタの実行が行われたりすると，エラーが引き起こされてしまうだろう。

7.2　クラス専用の記憶管理

これまでに示したnewとdeleteの実装は多目的のものであり，標準のバージョンのように，すべての場合に，すべての型に対して使用できることを前提としている。しかし，型あるいはアプリケーション特有の知識を活かすことで速度を改善したり，記憶管理を非常に簡潔にしたりすることが可能である。標準の記憶管理演算子を残したまま，こうした独自の情報を活用できるように，C++では，各クラス専用のバージョンの“operator new”と“operator delete”を定義することができるようになっている。クラス定義の中で宣言された場合，“operator new”と“operator delete”は静的メンバ関数になる。このため，これらの演算子には暗黙のthisポインタがなく，実行するにあたってオブジェクトを使う必要はない。これらの演算子が所定のクラス型のオブジェクトを確保したり，解放したりするためのものであることを考えれば，これは当然である。

たとえば，クラスcomplexの実装は非常に小規模かつ簡単である。こうした実装特有の細かい特徴を利用したnewとdelete演算子をメンバとして定義すると，記憶管理を相当改善することができる。

```
class complex {
   double re, im;
  public:
```

```
    void *operator new( size_t );
    void operator delete( void * );
    // ...
};
```

この“complex::operator new”と“complex::operator delete”は，クラスcomplexのオブジェクトの割り当てと削除のたびに呼び出される。オブジェクトの割り当てと削除がクラスcomplexのスコープの中で行われたかどうかと，これは無関係である。

```
#include <complex.h>

main() {
    complex *cp = new complex( 12 );
    int *ip = new int;
    *ip = 12;
    complex cgross = *cp * *ip;
}
```

ここで，ポインタcpは“complex::operator new”が確保した記憶領域を指し，ポインタipは標準の“operator new”が確保した記憶領域を指す。オブジェクトcgrossは自動オブジェクトであり，初期設定に先立ってnewを呼び出す必要はない。

以下では，各オブジェクトに静的配列の要素を割り当てることで，クラスcomplexの記憶管理を実装することにする。また，解放された要素は空きリストで管理する。“complex::operator new”は，最初に空きリストから(以前に割り当てられ)現在解放されている要素があるかどうかを調べる。空きリストに要素がなく，かつまだ割り当てられていない配列要素があれば，それを割り当てる。さもなければ，“complex::operator new”は失敗してヌルを返す。

```
static const int MAX = 100;

static class Rep {
    static Rep *free;
    static int num_used;
    union {
    double store[2];
    Rep *next;
    };
    friend class complex;
} mem[MAX];
```

```
void *
complex::operator new( size_t ) {
   if( Rep::free ) {
      Rep *tmp = Rep::free;
      Rep::free = Rep::free->next;
      return tmp->store;
   }
   else if( Rep::num_used < MAX )
      return mem[ Rep::num_used++ ].store;
   else
      return 0;
}
```

“complex::operator delete”は，解放しようとする配列要素を空きリストの先頭に追加する。

```
void
complex::operator delete( void *p ) {
   ((Rep *)p)->next = Rep::free;
   Rep::free = (Rep *)p;
}
```

ここで，“complex::operator new”は引数の値を使っていないが，常にクラスcomplexのオブジェクトを格納できるだけの記憶領域を返すことに注意してほしい。これは，クラスcomplexからクラスが派生されることはないという暗黙の仮定によるものである。こうした仮定がよいときもあるが，以下の例のように，ときとして問題が生じることもあるだろう。

```
class Point3 : public complex {
   // 三次元空間上のある点を表現する
   double z;
  public:
   // ...
};
Point3 *p = new Point3;
```

ここで示したクラスPoint3は，クラスcomplexのメンバとともにnewとdeleteの実装も継承するため，Point3のコンストラクタの呼び出しは“complex::operator new”を呼び出す。この“complex::operator new”はcomplexには十分であるが，Point3に不足する大きさの記憶領域を返す。

クラス専用のnewとdeleteを定義するにあたって，各型の性質だけを考えること

から離れ，さらに視野を拡張し，アプリケーション全体の性質に依存する記憶管理機構をデザインすることもできる。

第5章で，電卓プログラムのためにデザインしたNode階層を考えよう。ここでは，この電卓プログラムが一度に作成する抽象構文木は1つだけであり，この構文木は評価後，木全体が一度に削除されると仮定しよう。この条件が与えられれば，効率を改善すべく記憶管理機構を再びデザインし直すことができる。

```
class Node {
  protected:
   Node() {}
  public:
   // ...
   void *operator new( size_t );
   void operator delete( void * ) {}
   static void flush(); // すべてのメモリを回収する。
   enum { SZ = 1000 };
   class Buf {
      Buf *next;
      char mem[SZ];
      friend class Node;
   };
  private:
   static Buf b;          // 初期バッファ
   static Buf *bp;        // 現在のバッファへのポインタ
   static char *memp;     // バッファの現在位置へのポインタ
   // ...
};
```

“Node::operator new” と “Node::operator delete” は，Nodeから派生したすべてのクラスに継承される。このため，各派生クラスに専用のバージョンを用意するか，あるいはすべての節型に対して役目を果すことができる “Node::operator new” と “Node::operator delete” を用意する必要がある。ここでは後者を選ぶ。

```
void *
Node::operator new( size_t sz ) {
   if( memp+sz > &bp->mem[SZ] )
      if( bp->next ) {
         bp = bp->next;
         memp = bp->mem;
      }
      else if( bp = bp->next = new Buf )
         memp = bp->mem;
      else
```

```
        return 0;
    char *r = memp;
    memp += sz;
    return r;
}
```

最初は，アプリケーションの実行時に，たいていの場合には十分であろうと考えられる大きさの静的バッファを用意し，必要に応じてフリーストア領域に追加のバッファを確保する。なお，バッファ自体はNode 型でもNode からの派生クラス型でもないので，追加のバッファを確保するためのnew には，標準の“operator new”が使われる。

“operator delete”の実装は，delete が使われるときには常に抽象構文木すべてが解放されるという事実を利用している。このため，“operator delete”は何も行わず，代わりに静的メンバ関数flush がすべてのメモリを解放する。解放されたメモリ(追加で確保されたメモリも)は，“Node::operator new”の管理下に戻され，すべて保持され続け，次にNode 木が構築されるときに再利用される。

```
Node::Buf Node::b;
Node::Buf *Node::bp = &b;
char *Node::memp = b.mem;

void
Node::flush()
    { bp = &b; memp = b.mem; }

// ...
int
limited_use() {
    // -11+21*93 を評価する。
    Node *np =
        new Plus(
            // ...
    );
    int result = np->eval();
    delete np;       // デストラクタが実行される。
    Node::flush(); // メモリは再利用される。
    return result;
}
```

このバージョンのnew とdelete は標準バージョンのものにくらべて高速であるが，特定のアプリケーションの抽象構文木の使用法と密接に結びついている。

new 演算子は，クラス中で多重定義されてもよい。たとえば，独特の特性を有する

数種のメモリ領域にNodeオブジェクトを割り当てたいとしよう。

```
class Arena;

class Node {
  public:
    // 例題7-4参照
    void *operator new( size_t );
    void *operator new( size_t, Arena * );
    void operator delete( void * );
    // ...
};
// ...
extern Arena *ephemeral;      // 一時領域
extern Arena *persistent;     // 永続領域
extern Arena *shared;         // 共有領域
// ...
Node *np1 = new ( shared ) Int( 12 );
Node *np2 = new ( persistent ) Int( 2 );
Node *np3 = new Int( 3 );     // デフォルトの領域
```

“operator delete”は単一の実装で，多重定義された“operator new”の複数の実装によって割り当てられた記憶領域を削除できる必要がある。

クラス専用のnewとdeleteは，特定のクラス型(と，明示的にオーバライドされている場合を除けば，このクラス型から派生したクラス型)のオブジェクトの確保と解放の意味にまでも関係を及ぼす。たとえば，ある型の正しい割り当ての意味として，オブジェクトをフリーストア上に割り当てることをすべて禁止するというものが考えられる。

ここで，資源コントローラ型のデザインを考えよう。ここでは，(ファイルハンドル，データベースロック，共有メモリ領域などの)資源の割り当てと解放を制御するための資源「ハンドル」を実行時スタック上に確保することにする。この実装は，資源の制御のために，コンストラクタとデストラクタの意味論を利用している。資源ハンドルの初期設定には資源の割り当てという副作用があり，資源ハンドルの破棄には資源の解放という副作用がある。資源ハンドルが実行時に自動オブジェクトとして，あるいは静的オブジェクトとして割り当てられた場合，資源の解放も同様に自動的に行われる。

```
class ResourceHandle {
  protected:
    ResourceHandle();
```

```
  public:
   virtual ~ResourceHandle() = 0;
  protected:
   void *operator new( size_t ) { return 0; }
   void operator delete( void * ) {}
  private:
   // ...
};
```

ここで，抽象基底クラスを作成し，資源ハンドルの本質的な特質を隠してしまうことにする。ResourceHandleから派生したクラスは，資源を割り当てるためのコンストラクタと資源を解放するためのデストラクタを持ち，protected部の記憶管理演算子を継承する。これらの記憶管理演算子はprotectedであるため，ResourceHandleからの派生クラスのメンバとフレンドだけがこれらを使うことができる。したがって，ResourceHandleから派生したクラスをフリーストアに確保しようという試みはコンパイル時にエラーとして検出される。

```
class FileHandle : public ResourceHandle {
    FILE *fp;
  public:
   FileHandle( const char *filename,
      const char *filemode = "r" )
      : fp( fopen( filename, filemode ) ) {}
   ~FileHandle() { if( fp ) fclose( fp ); }
   operator FILE *() { return fp; }
};

FileHandle globfile( "cfront.h", "r+" );  // OK: 静的

ptrdiff_t
filediff( const Pathname &path ) {
   FileHandle fh( path );      // OK: 自動
   FileHandle *fhp =
      new FileHandle( path ); // エラー, フリーストア!
   return fh - *fhp;
}
```

すでに述べたように，クラス専用のnewとdeleteは，所定のクラス型のオブジェクトの記憶管理のためだけに実行され，たとえその演算子が定義されているクラスのスコープ中であっても，ほかの型のオブジェクトのためには使われない。このことはクラスオブジェクトの配列に対しても適用される。クラスオブジェクトの配列はクラスオブジェクトではないので，クラス専用の記憶管理演算子は，配列のために暗黙のう

ちには実行されない。

```
complex *vector = new complex[3];
```

この割り当てには，“complex::operator new”ではなく，“::operator new”が実行される[1]。

一般的に，とても単純で扱いやすいクラス型および組み込み型を除けば，配列は割り当てるべきでない。クラスの効果のほとんどは継承と動的束縛にあるので，オブジェクトの配列を使うと，これらの利用はひどく制限されてしまうだろう。通常は，クラスオブジェクトの配列の代わりにクラスオブジェクトへのポインタの配列が使われる。

```
Node *cses = new Node *[MAX];
```

これに加えて，コンパイラによっては，配列の要素に初期設定子を与える場合に制限のあることがある。配列の確保と解放に関する詳細は，C++コンパイラのドキュメントを参照されたい。

7.3 コピーの意味

自分のクラス型と同じ型の引数で呼び出されるコンストラクタは，オブジェクトを同じクラス型のオブジェクトで初期設定する方法を示している。自分のクラス型と同じ型の引数で実行することのできる代入演算子は，オブジェクトを同じクラス型のオブジェクトで初期設定する方法を示している。これら2つの演算は，所定のクラスの「コピーの意味」を規定する。クラスXに対する，これらの演算は，概して以下のように宣言される。

```
class X {
    // ...
    X( const X & );
    X &operator =( const X & );
    // ...
};
```

[1] 訳注：新しい標準規格案では，配列領域の記憶管理演算子が新たに定義された。付録B.5参照

利用者定義の「コピーの意味」の規定法はすでに例示してある。第4章のString型では，Stringがコピーされるときには常に基礎部分のString::Repオブジェクトの参照カウンタが正しい管理状態にあることを確実にするために，コンストラクタ“String(const String &)”と代入演算子“operator =(const String &)”を用意している。

コピーによる初期設定のデフォルトの実装は，あるオブジェクトでもう1つのオブジェクトをメンバ毎に初期設定するというものである。コピーによる代入のデフォルトの実装は，あるオブジェクトからもう1つのオブジェクトにメンバ毎に代入するというものである。このデフォルトの実装は，Stringオブジェクトにとっては，参照カウンタを正しく管理する必要があるので誤りである。しかしこのデフォルトの実装はcomplexオブジェクトにとってはふさわしいので，クラスcomplexには明示的に「コピーの意味」を与える必要はない。第4章のSortedCollection型は第3の可能性を示している。SortedCollection型の第2の実装では，デフォルトのメンバ毎の「コピーの意味」は間違った結果を生み出してしまうだろう。また，大きいコレクションのコピーは潜在的にあまりに高価であり，あまり使われないと思われる。これらの理由から，SortedCollection型では，コピー演算をクラスのprivateメンバとして宣言するというアプローチを取った。この結果，SorttedCollectionオブジェクトをコピーしようとしたところで，コンパイル時にprivateメンバの呼び出しとしてエラーが検出されるだろう。クラスに「コピーの意味」を定義することは，そのオブジェクトのコピー演算には注意を払う必要があり，デフォルトの「コピーの意味」では適切でないということを意味している。

デフォルトの「コピーの意味」はあくまでも「メンバ毎」のコピーであって，常に「ビット毎の」コピー(あるメモリ領域をもう1つのメモリ領域に単にコピーすること)ではないことに注意されたい。デフォルトのメンバ毎のコピーは，基底クラスあるいはメンバで定義された「コピーの意味」にも従うように行われる。

```
class Text {
   Text *next;
   String phrase;
  public:
   // ...
};
```

```
Text a( "Hello, World!" );
Text b( a );
a = b;
```

aによるbの初期設定ではStringメンバのコピーコンストラクタが，aへのbの代入では同じく代入演算子が，コピー演算の一部として実行される。「コピーの意味」が定義された基底クラスあるいはメンバを持つクラスの場合，ビット毎のコピーではオブジェクトを正しくコピーできないので，コンパイラはメンバ毎のコピー演算を行う**コピーコンストラクタ**と**代入演算子**を暗黙のうちに定義する。コンパイラによって与えられる演算は，以下のように定義されたのと同じ動作を実行する。

```
inline
Text::Text( const Text &t )
    : next( t.next ),      // ビット毎
    phrase( t.phrase )     // String( const String & )
    {}

inline
Text &Text::operator =( const Text &t ) {
    next = t.next;         // ビット毎
    phrase = t.phrase;     // operator =( const String & )
}
```

これと同じ理由から，以下に示す状況のとき，コンパイラはデフォルトコンストラクタとデストラクタを暗黙のうちに定義する。コンストラクタの宣言を明示的に持たず，デフォルトコンストラクタを持つ基底クラスあるいはメンバを持つクラスの場合，コンパイラは，基底クラス部分のオブジェクトおよびメンバに対してデフォルトコンストラクタを実行して初期設定を行うという，**デフォルトコンストラクタ**を暗黙のうちに定義する。また，クラスがデストラクタの宣言を明示的に持たず，デストラクタを持つ基底クラスあるいはメンバを持つ場合，コンパイラは，基底クラス部分のオブジェクトおよびメンバに対してデストラクタを実行するという，**デストラクタ**を暗黙のうちに定義する。

クラスcomplexのように，いくつかの場合には，ビット毎の「コピーの意味」を当てにしたり，コンパイラによって定義されるデフォルトの「コピーの意味」と，デフォルトコンストラクタとデストラクタを使ったりすることが正しいことがある。こうした場合にも，これらのデフォルトの動作がクラスの実装に対して正しいかどうかに注意し，クラスの実装に何らかの修正があった場合，再度調べ直すようにすべきである。

しかし，コンパイラによってクラスに与えられたデフォルトの「コピーの意味」と，デフォルトコンストラクタとデストラクタが適切であることは稀であり，一般的にはこれらを明示的に指定すべきである。

7.4　一時オブジェクトと効率

第4章で示したStringクラスは，単一引数のコンストラクタを持ち，このコンストラクタを“char *”からStringへの型変換に使うことができた。この型変換は，必要とされるときに暗黙のうちに実行される。

```
class String {
   // ...
   String( const char * );
   friend int
      operator ==( const String &, const String & );
   // ...
};
// ...
String logfile;
//...
if( logfile == "/etc/accesses" )
   // ...
```

ifの条件式では，コンパイラは==演算子のオペランドの型をマッチさせるため，右辺の引数("/etc/accesses")から，Stringの単一引数コンストラクタを使って，String型の一時オブジェクトを作成し，初期設定する必要がある。この一時オブジェクトは，“operator ==”の第2仮引数“const String &”を初期設定するのに使われる。この式が終わったところで，この一時オブジェクトはStringデストラクタによって破壊される。

この動作は正しいのだが，効率を考えると，多くの型変換と付随する初期設定と破壊が問題視されるだろう。特定の関数呼び出しの効率を注目すれば，さらに多重定義関数を追加することで，型変換による一時オブジェクトの作成を回避できる。

```
class String {
   // ...
   friend int
      operator ==( const String &, const String & );
   friend int
      operator ==( const String &, const char * );
```

```
    friend int
        operator ==( const char *, const String & );
    // ...
};
```

これで，引数がStringと“char *”の組み合わせの場合，いずれかの演算子関数と厳密にマッチし，一時オブジェクトの作成は回避されるようになる。ただし，一般的には，単純でわかりやすいインターフェイスのほうが効率上のわずかの問題よりも優先される。

C++コンパイラ自体が，高効率という名声を得るために一時オブジェクトの生成を回避するための何らかの処理を行うこともあるだろう。以下の宣言を考えよう。

```
String notatemp = String( "Am I a temp?" );
```

この宣言の表面上の意味は，最初にString 型の一時オブジェクトを作成し，notatempの初期設定のために“String(const String &)”コンストラクタを呼び出すというものである。しかし実際には，C++コンパイラが一時オブジェクトの生成を避けるために，文字列で単にnotatempを初期設定するというのは正当であり，しばしば行われる。

仮引数の初期設定と関数の戻り値では，こうした型変換が行われることも多い。仮引数の初期設定の場合を考えてみよう。いくつかの場合，実引数を仮引数の正しい型に型変換するために，コンストラクタあるいは型変換演算子が実行される。この際に仮引数に値をコピーする前に，コピーする値を作成するために，呼び出し側に一時オブジェクトを作成する必要が生ずることがある。

```
void f( complex );
f( 12 );
```

整数12からcomplexへの型変換では，呼び出し側にcomplex型の一時オブジェクトが作成され，クラスcomplexのコンストラクタによって初期設定される。一時オブジェクトは仮引数へコピーされる。しかしコンパイラは，一時オブジェクトの作成を避けるために，呼び出された関数側で，complexを構成する目的で整数の実引数を使うようにしてもよい。

コンパイラはさらに凝った処理を適用してもよい。一時オブジェクトの初期設定あるいは破壊の結果としての副作用は，もしかするとコンパイラによって変更あるいは省略されてしまう可能性があるので，プログラマはこれに何ら依存すべきではない。

7.5　演算子->

記憶管理では，記憶領域の確保と解放のほかに記憶領域へのアクセスを扱う必要がある。また，アプリケーションは，記憶領域へのアクセスの前後に何らかの処理を行うコードを必要とすることが多い。

```
前処理
記憶領域へのアクセス
後処理
```

これらのコードは自動的に挿入されるわけではないので，うっかり忘れてしまうこともあれば,「楽天主義」により故意に無視されることもある。

```
class Info {
   void f();
   // ...
};

Info *p; // ポインタ p が常に非ヌルであることを調べるべきである
// ...
void
process_Info() {
   // ここでは，ポインタ p がヌルであることはまれである
   p->f();
   // ...
}
```

しかし，このprocess_Infoのように，思い付きの「手抜き」コードは悲劇をもたらすだろう。

こういった問題に対処するために，C++では演算子->を多重定義し，検査とアクセスという2つの意味を持った**高機能ポインタ**(smart pointer)を作成することができるようになっている。また，“operator ->”は，演算子()，[]，=と同じようにメンバ関数であり，さらに引数を取らず，“operator ->”が定義されているクラスのオブジェクト，ポインタあるいはリファレンスを返すように宣言される必要がある。

組み込みの演算子->は二項演算子であるが，利用者定義の演算子->は単項演算子として宣言されることに注意してほしい。したがって，利用者定義の“operator ->”は，直接的にであれ間接的にであれ，オブジェクトへのポインタを生成し，組み込みの演算子->が，この生成されたポインタを使ってクラスのメンバにアクセスするわけである。

```
class InfoPtr {
    static Info null_info;
    Info *p;
  public:
    InfoPtr( Info *ip = 0 ) : p( ip ) {}
    Info *operator ->() { return p ? p : &null_info; }
};
```

これは先程の問題の1つの解答であり，クラスInfoPtrは参照時に決してヌルポインタ値を返さない高機能ポインタ型である。しかも，誤ったアドレスに至る可能性がある++あるいは+といったポインタ演算も許されない。

```
InfoPtr p;
// ...
void
process_Info() {
    p->f();  // 安全
    // ...
}
```

関数process_Infoでの“p->f”という関数呼び出しは，pの“InfoPtr::operator ->”を呼び出し，“Info *”型の値を生成する。この“Info *”の値は，メンバInfo::fを呼び出すために使われる。

“operator ->”は，この演算子が定義されているクラスを返すように宣言することもできる。

```
class InfoLog {
    InfoPtr &p;
  public:
    InfoLog( InfoPtr &ip ) : p( ip ) {}
    InfoPtr &operator ->() {
        fprintf( logfile, "accessing an InfoPtr" );
        return p;
    }
};
```

このInfoLogの動作は，アクセスの直前にログ情報を保存することを除けば，InfoPtrと同じである。

```
extern InfoPtr p;
InfoLog wp = p;
// ...
wp->f();
```

今度の“wp->f”という関数呼び出しでは，最初に“InfoLog::operator ->”が呼び出され，ログ情報を保存した後でInfoPtrを返す。さらに，この戻り値を使って“InfoPtr::operator ->”が呼び出され(“InfoPtr::operator ->”は“Info *”の値を返す)，最後に“Info *”型の値を使って，Info::fが呼び出される。

高機能ポインタ型を使った，もっと現実的な例を考えてみよう。ディスク上にレコードファイルを配置し，(システム依存のディスク読み書き関数に煩わされずに)これらのレコードをあたかもメモリ中の配列の要素であるかのように扱うことのできるアプリケーションを書きたいとしよう。扱うレコードはたくさんあるため，すべてを単純にメモリ中に読み出すことはできない。しかし，ここではアプリケーションが同時に扱うレコード数は比較的少数であり，変更されるレコードはさらにわずかであるものとする。ディスク上のレコードへの多数の読み書きがあった場合に発生するオーバヘッドについては考慮しない。

この問題に対して2つの型を用意する。1つはこのデータベースのレコードファイルを表現する型Fileであり，もう1つはレコードファイルへの高機能ポインタ型Ptrである。

```
struct rec {    // ディスク上のレコードを表現する構造体の定義
    int key;
    int value;
    // ...
};

class File {
    FILE *fp;
    Sary *hd;
    rec *operator [](int);
  public:
    File( const char * );
```

```
   ~File();
   friend class Ptr;
};
```

利用者は，オープンしようとするファイルのパス名を与えて，物理的なファイルに対応するFile型のオブジェクトを生成する。File型で使われるFILE型と，ファイルのオープン，クローズ，読み書きのための関数はシステム依存である。ここでは標準入出力ライブラリstdioのものを使う。

```
File::File( const char *fname )
   : fp( fopen( fname, "r+" ) ), hd( 0 ) {}
```

比較的少数のレコードだけが読み書きされるので，ファイルレコード型Fileを，各要素をリスト構造で保持する疎配列として実装する。この疎配列は，インデックスとレコードの組による(未整列の)リストで表現される。

```
class Sary {
   Sary *next;
   int index;
   rec *p;
   Sary( int, Sary *, FILE * );
   friend class File;
};
```

このように，File型はシステム依存のファイル記述子fpと，レコードファイルの疎配列Saryから構成される。また“File::operator []”は，配列の要素を参照する演算子関数である。もし，参照された要素がメモリ中に不在であれば，レコードをディスクから読み出し，配列の新しい要素を生成する。

```
rec *
File::operator []( int i ) {
   for( Sary *sp = hd; sp; sp = sp->next )
      if( sp->index == i )
         break;
   if( !sp )
      sp = hd = new Sary( i, hd, fp );
   return sp->p;
}
```

配列の要素のコンストラクタは，リストの節の領域を確保し，ディスクからレコードの内容を読み出す。また，あとで要素の内容が変更されたかどうかを比較検査する

ために，読み出したレコードの複製を保存しておく。ディスク上のレコードは，変更されない限り更新されることはない。

```
Sary::Sary( int i, Sary *n, FILE *fp )
    : next( n ), index( i ) {
    const int sz = sizeof( rec );
    p = (rec *) new char[2*sz];
    fseek( fp, i*sz, 0 );
    fread( (char *)p, sz, 1, fp );
    *(p+1) = *p;
}
```

File型のデストラクタは，変更されたレコードの内容をディスク上のレコードに反映させ，それぞれの配列要素を削除し，レコードファイルをクローズする。

```
File::~File() {
    const int sz = sizeof(rec);
    while( hd ) {
        if( memcmp(sp->p, sp->p+1, sz) != 0 ) {
            fseek( fp, sp->index*sz, 0 );
            fwrite( (char *)sp->p, sz, 1, fp );
        }
        Sary *tmp = hd;
        hd = hd->next;
        delete tmp;
    }
    fclose( fp );
}
```

次にデータベースレコードFileへの利用者インターフェイスとして，高機能ポインタ型を用意する。

```
class Ptr {
    int index;
    File *fp;
  public:
    Ptr( File &f, int i = 0 )
        : fp(&f), index(i) {}
    rec *operator ->();
    rec &operator [](int);
    rec &operator *();
    friend Ptr operator +(Ptr, int);
    friend Ptr operator +(int, Ptr);
    friend ptrdiff_t operator -(Ptr, Ptr);
    friend Ptr operator -(Ptr, int);
    Ptr &operator ++();       // 前置
```

```
    Ptr &operator --();
    Ptr &operator ++(int);  // 後置
    Ptr &operator --(int);
    Ptr &operator +=(int);
    Ptr &operator -=(int);
    friend int operator ==(Ptr, Ptr);
    friend int operator !=(Ptr, Ptr);
    friend int operator <(Ptr, Ptr);
    friend int operator <=(Ptr, Ptr);
    friend int operator >(Ptr, Ptr);
    friend int operator >=(Ptr, Ptr);
};
```

ここでは，Fileをレコードの配列と考えることで，高機能ポインタPtrをFileへのポインタとして実装した。また，Ptrには，通常のポインタとしてのすべての算術演算が用意される。以下に，いくつかの実装例を示す。

```
Ptr &
Ptr::operator ++() {
    index++;
    return *this;
}

int
operator ==( Ptr a, Ptr b )
    { return (a.index == b.index) && (a.fp == b.fp); }
```

ほかの演算子の実装も同様である。このポインタによる算術演算では，Ptrのメンバであるファイルレコードへのインデックスindexをもとに演算が行われ，必要とされるまではFileへの参照は行われない。

レコードへの参照は，実際には“File::operator []”が行う。したがって，この高機能ポインタは，単純にPtr::indexに対応するFileの配列要素を返すだけである。

```
rec *
Ptr::operator ->()
    { return (*fp)[index]; }
```

C++の組み込みのポインタ型の演算子->，*，[]において，p->mは(*p).mと等価であり，*(p+i)はp[i]と等価であるという対応関係がある。したがって，利用者定義のポインタ演算に，同じ等価性を期待するのであれば，ほかの言語定義の等価性(たとえば，++と“+= 1”)と一緒に，明示的にこれらの等価性も定義する必要がある。

```
rec &
Ptr::operator *()
   { return *(*fp)[index]; }

rec &
Ptr::operator []( int i )
   { return *(*fp)[index+i]; }
```

これで，アプリケーションを作成する準備が整った。ここでは，すでにディスク上にデータベースが構築されていると仮定しよう。以下に，あるレコードの値を見つけて変更する例を示す。

```
void
alter( int key, const char *file, int num_recs, int val ) {
   File f = file;
   Ptr mid = f;
   Ptr low(f,0), high(f,num_recs-1);
   while( low <= high ) {
      mid = low + (high-low)/2;
      if( key < mid->key )
         high = mid - 1;
      else if( key > mid->key )
         low = mid + 1;
      else {
         mid->value = val;
         break;
      }
   }
}
```

関数alterは，fの宣言によって(引数fileで指定される)ファイルをオープンし，ファイルレコードの配列を作成する。さらにファイルレコードを二分探索するために，fへの3つの高機能ポインタlow，mid，highを作成している。引数keyに相当するレコードが見つかったら，対応する値を変更する。またfのデストラクタは，ファイルレコードの疎配列の後始末をし，変更のあった要素のみをディスク上のレコードに反映させる。

演　習

演習7-1　complex型に仮想関数を定義することで与えられる効果を考えよ。仮想関

数は，本章で示したcomplex型の効率のいい記憶割り当て手法にどういった効果を与えるだろうか。

演習 7-2 クラスcomplexの演算子newとdeleteの実装を変更して，任意のサイズの記憶領域の要求に応じられるようにせよ。

演習 7-3 クラス専用の新しいnewとdelete演算子を考えよう。ここでは「削除不可」を指定することで，特製の記憶領域を割り当てることができるものを作成してほしい。「削除不可」の記憶領域とは，deleteを実行しても，領域が削除されず残ることを意味する。

演習 7-4 本章で述べたNodeクラスに，“operator delete”と“operator new(size_t)”と“operator new(size_t, Arena *)”を実装せよ。

演習 7-5 共有メモリ領域中にオブジェクトを割り当てるためのものとして，演習 7-4のメモリ領域Arenaの実体を作成せよ。

演習 7-6 演習 4-28(コレクションへの述語論理による添字付け)の解法を再考しよう。確保した記憶領域はすべて，正しく再利用されているだろうか。実装は柔軟で，効率が良いだろうか。各自の解法で，以下の関数を試してみよ。

```
extern int moe( Etype );
extern int larry( Etype );
extern int curly( Etype );

void
test( SortedCollection &c ) {
    SortedCollection d;
    d = c[ moe ];
    d = c[ moe ][ larry ][ curly ];
}
```

演習 7-7 本章の“operator new(size_t, const String &)”中では，記憶領域の割り当てに，どうして

```
char *p = new char[n];
```

の代わりに

```
char *p = ::operator new( n );
```

という構文を使っているのだろうか。

演習 7-8[†]　本章のResourceHandleを実装せよ。

演習 7-9　本章で解説した機能を使って，演習 4-21 の二分木用の通りがけ順の制御抽象を再実装せよ。特に，代入と初期設定の意味に注意せよ。また，性能向上のため，クラス専用の記憶管理を与えよ。

演習 7-10[†]　フリーストア領域だけを指すことを保証された，高機能ポインタの総称型をデザインせよ。

演習 7-11　一部のプログラミング言語はオフセットをサポートしている。オフセットとは，ポインタに類似の機能で，簡単にいってしまえば，あるメモリ領域へのインデックスである。このメモリ領域はメモリ場とも呼ばれる。オフセットの第1の強みは，メモリ場をまるごと外部記憶に書き出し，後でほかのアドレスに読み戻すことができることにある。この際に，オフセット値の修正は不要である。

ある特定の型へのオフセットをデザインせよ。このとき，この型のオブジェクトをメモリ場中への割り当てたり，外部記憶に書き出したり，読み戻したりできるようにせよ。また，オフセットの実装をテンプレートを使って拡張せよ。

演習 7-12　複数のメモリ場を扱うことができるように，演習 7-11 のオフセットの実装を拡張せよ。

演習 7-13　範囲検査を行うように，本章のFileとPtr型を再実装せよ。

演習 7-14[†]　“operator ->”を使って，所定のオブジェクトの一部の演算をほかのオブジェクトに委譲する手段を示せ。また，この手段と継承を比較せよ。

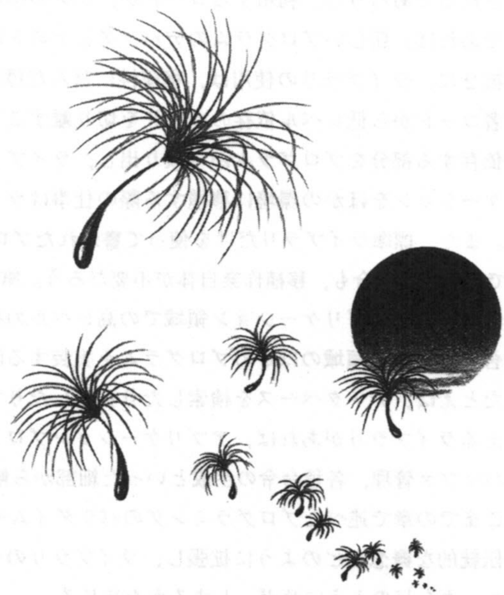
演習 7-15[†]　所定のクラスを自動オブジェクトあるいは静的オブジェクトとして作成することを禁止するテクニックを考え出せ。

演習 7-16　「未来」型を作成するためのテクニックを考え出せ。ここでの「未来」型とは，初期設定あるいは代入の処理から独立した実行スレッドで，意味のある値を計

算するものである。また，結果の値の参照は，計算が完了するまで禁止される。どのようにすれば，このテクニックを使って，アーキテクチャ独立の細粒度並列性を実装することができるか示せ。また，各自のテクニックを使って，演習 3-6 の永続ハッシュ表を基本にして「未来ハッシュ表」を作成せよ。

8

ライブラリ



ソフトウェアライブラリは，個々のプログラムから独立したモジュールの集まりとして構成されている。モジュールをプログラムから分離し，ライブラリにまとめることで，あらゆるプログラムからそれらを利用することができるようになる。伝統的な手続き型プログラミング環境では，ソフトウェアライブラリは，特定の機能を提供するための関数とデータ構造の集まりとして構成されている。具体的には，数学関数，オペレーティングシステムの基本機能へのアクセス関数，標準ファイルフォーマットのデータ構造の規定，標準入出力関数といったライブラリがある。

プログラム作成にライブラリを使うことにはいくつかの長所があるが，特に以下の3つが重要である。第1に，ライブラリの利用によりアプリケーションは単一のソースコードを源とするコードを使うことができる。アプリケーションが必要とする機能が既にライブラリに与えられている場合，アプリケーション開発の仕事の一部は既に終わっていると考えることができるので，アプリケーションのデザインとコード作成にかかる時間は短縮されるであろうし，利用するコードが，バグの枯れた，安定して正しく動作するものであれば，新しいプログラムのデバッグとテストにかかる時間も短縮されるだろう。第2に，ライブラリの使用は，移植性に富んだ標準のインターフェイスを与え，利用者コードから低レベル依存のコードを切り離すことができるようにする。実行環境に依存する部分をプログラムから取り出し，ライブラリに閉じ込めてしまえば，アプリケーションをほかの環境に移植する際の仕事はライブラリを移植することだけである。また，標準ライブラリだけを使って書かれたプログラムであれば，ほかの多くの環境で実行する場合も，移植作業自体が不要だろう。第3に，ライブラリのインターフェイスが特定のアプリケーション領域での高レベルの抽象概念からデザインされている場合には，同じ領域のほかのプログラムを実装する際の苦労を軽減することができる。たとえば，データベースを検索したり更新したりするための高レベルの抽象概念を与えるライブラリがあれば，アプリケーションプログラマは，データベースとの接続，バッファ管理，各種命令の生成といった細部から解放される。

この章では，ここまでの章で述べたプログラミングのパラダイムやC++言語の特徴が，ライブラリの伝統的な概念をどのように拡張し，ライブラリのデザインと実装のための新しいテクニックをどのようにサポートするかを論じる。

8.1 インターフェイスの再構築

第4章で論じた抽象データ型のデザイン方法は，ライブラリのインターフェイスにも応用できる。C++のクラスを使えば，データを隠蔽し，関連する関数群を1つの概念単位にまとめることによって，ライブラリのインターフェイスをわかりやすくすることができる。たとえば，既に存在するライブラリをクラスに閉じ込めてしまい，代わりにライブラリ本来の基本的な抽象概念を表すインターフェイスを与えることもできるし，さらに発展させて，クラスを低レベルの既存機能から高レベルの抽象概念を構築するための機構として使うこともできる。

Cの標準入出力ライブラリstdioが提供している入出力関数を例にとろう。

```
FILE *fopen( const char *, const char * );
int fclose( FILE * );
int fflush( FILE * );
int fprintf( FILE *, const char *, ... );
int fscanf( FILE *, const char *, ... );
```

これらの関数は，ファイルオブジェクトに対する演算の集まりと考えることができる。ここでfopenはコンストラクタ，fcloseはデストラクタの役割を果たし，ほかの関数は，暗黙の引数thisを第1引数に持つと考えれば，ちょうどメンバ関数である。

この抽象概念を明らかにするために，クラスを使って次のインターフェイスを定義しよう。

```
class File {
   FILE * const f;
  public:
   File( const char *path, const char *mode = "w" )
      : f( fopen( path, mode ) ) {}
   ~File() { fclose( f ); }
   int flush() { return fflush( f ); }
   int printf( const char *, ... );
   int scanf( const char *, ...);
   // ...
};
```

File::printfやFile::scanfの実装は，最後の引数が可変個であるため，すこし複雑である。これは，メンバ関数printfとscanfの仮引数を，stdioのfprintfやfscanfを呼び出す際の実引数として使いたいと考えたからである。しかし，省略

符号(...)が示すように，実行時まで引数の数を決めることができない。この条件を対処する手段はいくつかあるが，ほとんどの手段は環境依存である。以下の例では，File::printfの実装を簡単にする目的で，標準ヘッダファイルstdarg.hが提供する“varargs”(**variable arguments**，可変個の引数)機能を使用した。引数並び型va_listのオブジェクトapは，引数並びへのアクセス法に関する情報を保持するためのものである。最初に，va_start関数が省略符号(...)の直前に宣言されている引数fmtによってapを初期設定している。va_arg関数は，引数並びから第2引数で指定された型の引数を順番に取り出して返す。最後に，varargs機能を正常に終了させるためにva_end関数が呼び出される。

```
int
File::printf( const char *fmt, ... ) {
   va_list ap;
   va_start( ap, fmt );

   register int c;
   while( c = *fmt++ ) {
      if( c == '%' ) {
         switch( c = *fmt++ ) {
         case 'c':    // 文字
            fprintf( f, "%c", va_arg( ap, int ) );
            break;
         case 'd':    // 10進整数
            fprintf( f, "%d", va_arg( ap, int ) );
            break;
         case 'g':    // 浮動小数点数
            fprintf( f, "%g", va_arg( ap, double ) );
            break;
         case 's':    // 文字列
            fprintf( f, "%s", va_arg( ap, char * ) );
            break;
         // etc...
         }
      }
      else
         fprintf( f, "%c", c );
   }

   va_end( ap );
}
```

これで，ライブラリstdioを使うために，Fileオブジェクトを作成し，使うことができる。また，この例では，あらかじめ定義されている標準入出力ファイルstdout,

stdinに対する入出力関数printf，scanfと名前を同じにするために，出力のためのメンバ関数名をfprintfからprintfに，入力のためのメンバ関数名をfscanfからscanfに変更している。こうすれば，ライブラリ関数printfとscanfを，Fileオブジェクトが省略された場合の入出力と考えることもできる。

```
void
tee( const char *fn ) {
    File f = fn;
    char c;
    while( scanf( "%c", &c ) != EOF ) {
        f.printf( "%c", c );
        printf( "%c", c );
    }
}
```

クラスFileにライブラリstdioを閉じ込めることで，ファイルという抽象概念を表すインターフェイスを与えたわけであるが，インターフェイスの抽象レベル自体は古いライブラリのものと同じである。ここで，標準ファイル入出力の機能をほかのクラスの中に閉じ込め，さらに高いレベルでの抽象概念を構築することで，利用者プログラムが処理する必要のある細部を大幅に削減することもできる。たとえば，以下のTextInputクラスは，入力ファイルに対する処理すべてを管理するためのもので，入力ファイルのオープン，クローズ，入力テキストのすべての行に対する繰り返し処理，各入力行を処理する際に利用者コード側で使われる動的に割り当てられたStringオブジェクトの管理といった機能を持つ。

```
class String;
class File;

typedef void (*LineHandler)( String & );

class TextInput {
    class StringManager &manager;
    File &textfile;
     public:
    void do_processing( LineHandler linehandler );
        // do_processingは，利用者コードのlinehandlerに
        // 渡されるStringオブジェクトを作成する。
        // これらのオブジェクトはTextInputが管理する。
    int is_open() const;
```

```
    TextInput( const char *path );
    ~TextInput();
        // do_processingが作成したすべての
        // Stringオブジェクトは, TextInput
        // の中で破壊される。
};
```

利用者コードの側からは，入力ファイルを行単位で処理するための行ハンドラ関数を与える。利用者コードが入力ファイル自体に行う一連の作業は非常に簡潔である。最初に，TextInput オブジェクトを作成する。次に入力ファイルのオープンに成功したかどうかを検査し，最後に，do_processing を実行するだけである。do_processing の実際の仕事は，引数として与える行ハンドラ関数が行う。

```
extern void one_line_at_a_time( String & );
extern error( const char *, ... );

main() {
    TextInput text = "textfile";
    if( text.is_open() )
        text.do_processing( one_line_at_a_time );
    else
        error( "Cannot open text" );
}
```

また，クラスTextInputの宣言では「注釈行」を使って，TextInputオブジェクトには自分が作成したStringオブジェクトの管理責任があることを明示的に書いてある。クラス宣言の一部として，この種のメモリ管理責任の所存を正式に示すことはできないが，クラスの使用法を正しく理解してもらうためには，こういった情報は，クラスインターフェイスの一部として不可欠である。したがって，利用者コードの側では，TextInputオブジェクトで作成されたStringへのポインタ(またはリファレンス)を，TextInputの削除後まで存続させてはならない。この種のポインタ(またはリファレンス)は不正オブジェクトを指す**ダンゲリングポインタ**をプログラムに残してしまう。したがって，TextInputオブジェクトの破棄後も入力行の複製を残したい場合には，利用者コードの側で，行ハンドラ関数に渡されたStringの複製を独自に作成する必要がある。

また，TextInputで作成されるStringの管理には2つの選択肢が考えられる。1つは，Stringオブジェクトは行ハンドラ関数を呼び出している間だけ存在するというものである。この場合，TextInput::do_processingがStringオブジェクト

の管理責任を負い，Stringオブジェクトは，実際には次のように自動オブジェクトとして作成されるだろう。

```
void
TextInput::do_processing( LineHandler linehandler ) {
    // ...
    {
        String line( linebuffer );
        linehandler( line );
    }
    // ...
}
```

行ハンドラ関数linehandlerの終了後も入力行を残すには，利用者コードの側で独自の複製を作成すればよい。ただし，ここで複製したオブジェクトの管理責任は利用者プログラムにある。

もう1つの選択肢は，Stringを「削除する責任」をすべて利用者コードにまかすというものである。しかし，この選択肢では，利用者コードがクラスを正しく使う責任を軽視した場合には，利用できるメモリを使い切ってしまうというエラーの可能性がある。したがって，こうした利用者コードの責任は，ドキュメントと，できればインターフェイスデザインの両者でしっかりと強調すべきである。これを実現する苦肉の策として，行ハンドラ関数への引数をポインタに変更することができる。ポインタは，動的に割り当てられたオブジェクトへアクセスする標準手段である。もちろん以下のように，メモリ管理法は注釈行に明示的に書くべきである。

```
typedef void (*LineHandler)( String * );

class TextInput {
    // ...
    void do_processing( LineHandler linehandler );
        // 警告
        // do_processingは，利用者コードのlinehandleに
        // 渡されるStringオブジェクトを作成する。
        // 注意： これらのオブジェクトを削除する
        // 責任は利用者コードにある。
    // ...
};
```

ライブラリが，高レベルの抽象概念で，問題領域すべてに渡って機能を与えている場合，ライブラリと利用者コード間で受け渡しされる情報には，これらの情報の抽象概念

を表現するオブジェクトが使われることがある。この場合，ライブラリのインターフェイスとして使われるオブジェクトの寿命をどのように管理するかという問題をライブラリのデザイン時に処理する必要がある。ライブラリのデザイン時のオブジェクト管理法の決定は，ライブラリを簡単に使うことができるかどうかだけではなく，ライブラリが正しく使われるかどうかにさえも関係してくることがある。また，こうした決定は，ライブラリの実装と利用者コードの両者の効率に対しても，深く関係することがある。

クラスに既存のライブラリを閉じ込み，高レベルの抽象概念でライブラリインターフェイスを構築すれば，ライブラリへアクセスするプログラムコードを非常に簡潔に書ける。しかし，抽象インターフェイスによって利用者コードは細部を扱うことから解放されるのだが，いままでは利用者コードで対処していた正しさと効率に関する責任は，ライブラリのデザインと実装の問題へと移動しているということに注意されたい。

8.2　エラーのインターフェイスと例外処理

ライブラリは，利用者プログラムからの要求すべてに，常に対処できるわけではない。この原因は，利用者の間違いから環境状態の異常まで幅広く多様である。これらのいくつか場合，利用者プログラムの側で，ライブラリで起こったこうした失敗を検出し，適切に対処する必要がある。したがって，ライブラリには，エラー情報あるいは状態情報を利用者コード側に適切に報告するための手段を与えたほうがよい。こうした状態情報を与えるインターフェイスに，どういったものが適切であるかは，発生したエラーの悲惨さと異常さの程度に依る。

stdioの関数は，戻り値として状態情報を与える。fopen関数は，指定ファイルをオープンできなければ，ヌルポインタを返す。この失敗は，間違ったファイル名を引数として渡したといった利用者の間違いか，ファイルのディレクトリへのアクセスが制限されているといった環境状態の結果が考えられる。どちらにしても，この失敗は，標準のインターフェイスの一部で検出できるべきであるほど，頻繁に発生しうるものだと思われる。プログラムは，fopenの戻り値が正しいポインタであるかどうかを，stdioのほかの関数に渡す前に検査する必要がある。したがって，前節で示した，ライブラリstdioの新しいインターフェイスFileクラスの実装を完成させるためには，ファイルのオープンに成功したかどうかを検査する手段を与えたほうがよい。

```
inline int
File::valid() const { return f != 0; }
```

stdioライブラリには，エラーに対するもう1つのインターフェイスとして，大域変数errnoがある。ライブラリ関数からの戻り値が失敗を示す場合，errnoに失敗の原因に関する情報を示す整数のコードが設定される。errnoは，標準ライブラリの一部であるが，大域変数によるこのインターフェイスは貧弱である。何が原因でerrnoが設定されているかが，常に明らかであるわけではなく，またプログラマの多くはerrnoと，errnoに設定されるコード値が標準のインターフェイスの一部であることを忘れてしまっている。一般的には，大域変数は，単一のプロセス中で複数のスレッドを実行する場合には相互干渉の可能性を増加させてしまう。ただし，多重スレッド環境で使われる標準ライブラリの中には，安全にerrnoを使用できるように実装されているものがあると思われる。

前節のTextInputクラスでは，入力ファイルの状態を問うメンバ関数is_openを明示的に与えている。このクラスに，errnoによって与えられる，より細かい状態情報を返すほかの機能を与えることもできる。また，このクラスでstdioのエラー検査よりも厳しい検査を行ってもよい。たとえば，ファイルをオープンする前に正しいテキストファイルかどうかを検査するのもよいだろう。TextInput::do_processingは，テキストが正しい条件にあるかどうかにかかわらずうまく動作するように実装されているべきである。たとえば，入力ファイルが非オープンであるとか，バイナリファイルである場合には，この関数は何もしないようにするというのはどうだろう。高レベルの抽象概念としてのTextInputが，エラー状態に対し，容易に検出でき，ふさわしいデフォルトの動作があるにもかかわらず，プログラムを終了させてしまうのは好ましいこととは思えない。

特に，コンストラクタとデストラクタには，エラーの発生に対してしっかりとした動作が要求される。この根拠は，コンストラクタあるいはデストラクタから戻った後では，クラスの利用者がエラー状態を問うことが難しいことが多いことによるものである。たとえば，メンバあるいは基底クラスの部分オブジェクトを初期設定するために，あるコンストラクタの中から，ほかのコンストラクタが呼び出される場合もある。この場合，制御の流れが(初期設定のエラー状態を検査するであろう)コンストラクタ本体に戻る前に，さらに別のメンバまたは基底クラスの初期設定が行われることもあ

る。同様に，あるデストラクタでは，基底クラスあるいはメンバのデストラクタを呼び出されることもあるだろうが，この場合も基底クラスとメンバすべてのデストラクタの実行が完了するまで，エラーを検査することはできない。したがって，コンストラクタとデストラクタの内部において発生したエラーに利用者コードで対処するのは，実際には非常に難しい。

一般的に，オブジェクトは，たとえエラー状態にあったとしても，エラーが破滅的である場合を除けば，利用者が状態を問うまでは，使われ続けるというのがよい選択だと思われる。たとえば，コンストラクタがファイルのオープンに失敗した事態に対しても強固であるように，Fileのメンバ関数を再実装してみよう。

```
inline int
File::flush() { return f ? fflush( f ) : 0; }
```

TextInput::do_processingが処理に失敗する1つの例としては，動的メモリの確保に失敗することが考えられる。この失敗は，入力行を保持するオブジェクトを構築する際に，Stringライブラリのどこかで起こっていると思われる。このように，起こる頻度が低い失敗によって，ライブラリ関数の正常完了が妨害された場合，ライブラリは単に実行を中止し，次に何をするかを利用者プログラムにゆだねるのが適切であろう。

C++の例外処理機能は，エラー状態が検出されたところで，通常の制御の流れを，このエラー状態を正しく処理できるところに飛び移らせる機構を与える。何段階にも渡って深く呼び出されている可能性のあるライブラリ関数の内部から利用者コードへ飛び移るために，ライブラリコードはthrow式を実行する。一般的に，throw式には，throwを起こしたエラー状態を示すオブジェクトを引数として指定する。アプリケーションは，投げられた例外オブジェクトを，tryブロックと後続のコードによって解釈する。tryブロックは1つ以上の例外ハンドラを持ち，各例外ハンドラは，catch節と例外を処理するブロックから成る。ハンドラは，例外に対して，throw式の引数がcatch節の宣言にマッチした場合に実行される。

たとえばあるライブラリが，エラーのインターフェイスの一部として，数種の致命的なエラーを表現するクラス階層を持っていたとしよう。

```
class FatalError {
    char *msg;
  public:
```

```
        FatalError( const char *message );
        virtual void handleit()
            { log( msg ); exit( 1 ); }
    };

    class StackOverflow : public FatalError {
      public:
        StackOverflow()
            : FatalError( "stack overflow" ) {}
        void handleit() {
            if( ++count <= max ) {
                log( "stack overflow:  don't call so much" );
            else
                FatalError::handleit();
        }
      private:
        static long count;
        enum { max = 127 };
    };

    class MathErr : public FatalError {
      public:
        MathErr()
            : FatalError( "math error" ) {}
    };

    // ...
```

ライブラリコードは，回復できない状態を検出すると，FatalError型の派生クラス型のオブジェクトを投げる。

```
    if( cant_call )
        throw StackOverflow();
```

利用者プログラムでは，ライブラリ関数を呼び出す部分を，1つ以上の例外ハンドラが続くtryブロックで直接あるいは関接的に取り囲み，例外を受け止めることができるようにする。

```
    main() {
        while( continue_main_loop() ) {
            try {
                // ...
                something();
                // ...
            }
            catch( MathErr & ) {
```

```
            // 算術エラーは無視する
            // 再度，繰り返し処理
         }
         catch( FatalError &fatalerr ) {
            fatalerr.handleit();
         }
         catch( OtherLibError &othererr ) {
            // ...
         }
         // etc.
      }
      exit( 0 );
   }
```

ライブラリ内部でthrow式が実行されると，投げられた型にマッチするcatch節を持つtryブロックにぶつかるまで，実行スタックが解放される。この例では，StackOverflowオブジェクトからpublic基底クラスFatalErrorのリファレンスへの言語定義の型変換があるので，FatalErrorのcatch節にマッチする。このハンドラの実行後，ハンドラがプログラムを終了させたり，ほかの例外を投げたりする場合を除けば，制御の流れはtryブロックのハンドラ群の後へと続く。

例外処理に対応した最近のC++の処理系には，メモリの動的確保に失敗した際に投げる，標準例外型を持っているものもある。こういった環境では，TextInputを使う利用者コードで，メモリ確保の失敗をうまく報告するハンドラを設定することもできる。以下の例では，プログラムの終了前にエラーメッセージを表示するために，環境例外型xallocが受け止められる。

```
   if( text.is_open() )
      try {
         text.do_processing( one_line_at_a_time );
      }
      catch( xalloc &mem ) {
         error( "out of memory: processing incomplete" );
         exit( 1 );
      }
   else
      error( "Cannot open text" );
```

投げられた例外には，実行スタックを解放し，マッチしたハンドラを実行することのほかに，スタックを解放する結果として，スタックから削除されるオブジェクトのデ

ストラクタを実行するという副作用がある。

第7章のResourceHandle型を思い出してほしい。ResourceHandle型から派生した型のオブジェクトは，各自のコンストラクタとデストラクタを実行する副作用として資源の取得と解放を制御していた。たとえばファイルハンドル，メモリ場，データベースロック，プロセスといった資源は，スタック上の資源ハンドルを経由して取得と解放が行われるが，例外処理によってスタックが解放されるときにもこれらは正しく解放される。

例外処理は，プログラミングとデザインための機能として便利であるが，多重継承や演算子の多重定義と同様に，使われすぎることが多い。投げられた例外が誰にも受け止められなかったら，プログラムは終了してしまうため，例外はライブラリの回復できない失敗に対してだけ投げられるべきである。また，例外処理は高価であり，例外ハンドラの設定時にも，例外を投げるときにも，プログラムに多くのオーバヘッドを与えてしまうことがある。例外処理は，ライブラリが処理に失敗した際に，同じ処理を再実行したり，何かほかの処理を試したりする機会を与えることもできるし，プログラムが素直に終了してしまう前に，何らかの状況情報を表示する機能を与えることもできる。しかし，例外処理はあくまでも例外に対処するために使われるべきで，ありふれた状態レポートのために使われるべきではない。

8.3 調整できるライブラリ

ライブラリには，汎用のデータ構造を与えたり，制御の流れを構造化するための汎用の機構を与えたりするものもある。こういったライブラリは，プログラムのほかの部分と一緒に働けるように，利用者によって調整できなければならない。C++では，汎用のデータ構造はクラステンプレートを使って実装され，テンプレートは利用者プログラムで指定される型と定数用のインスタンスを生成することで調整される。汎用のデータ構造は第4章で解説したので，ここではこれ以上触れない。制御機構ライブラリの場合，利用者コードの側で，制御の枠組中で実行される動作を実装することで調整される。

8.1 節のTextInput クラスでは，利用者コードによって調整することができる制御機構をライブラリに与えるためのテクニックを披露した。関数TextInput::do_processingは，入力ファイルの各行に対する繰り返し構造を実現するが，この繰り

返しでは，指定動作として利用者コードで与えられたLineHandlerを実行する。このテクニックには「コールバック」として知られている。利用者コードの側では，コールバック関数をライブラリに登録し，ライブラリは，制御機構の中から利用者コードの関数をコールバックする。コールバック関数は，ライブラリにパラメタとして渡される関数へのポインタを経由して呼び出される。

継承もライブラリを調整するために使われる。ライブラリが基底クラスの集合を与え，利用者プログラムは，これらの基底クラスからの派生クラスを使う。調整と継承の例には，第6章で示したタスクライブラリがある。ライブラリのクラスtaskにコルーチンの制御機構を与え，taskからの派生クラスは，各コンストラクタ中で，利用者プログラム用のコルーチンを実装する。

継承と仮想関数は，コールバック関数を実現するもう1つの機構を与える。このテクニックでは，ライブラリのクラスで与えられる仮想関数にデフォルトの動作を与え，利用者コードは，これらのクラスをデフォルトのまま使ってもよいし，継承して，仮想関数をオーバライドすることで，動作を調整してもよい。

たとえば，GUI(Graphical User Interface)のボタンオブジェクトをC++のクラスで実装することを考えよう。

```
class Button : public Widget {
    // ...
    virtual void action();
    virtual void show_help_message() const;
    // ...
};
```

ここでは，Buttonは，GUIライブラリのクラスの1つであると仮定しよう。Widgetクラスは，このライブラリで利用できるすべてのGUI要素の基礎の実装である。Buttonのメンバ関数actionは，GUI上のボタンオブジェクトが「押された」ときに，ライブラリによって呼び出され，show_help_messageは「ヘルプ」を要求するイベントに対する応答時に呼び出される。要するに，Buttonはプロトコルを規定し，Buttonから派生するあらゆる型は，このプロトコルにしたがう。第5章のProtocol型と同じように，Buttonから派生したボタン型の利用者は，各派生クラスの詳細を気にする必要はなく，基底クラスのインターフェイスを使って，それらを扱うことができる。さらに，ボタンにプログラム固有の動作を与えるには，Buttonのデフォルト機能をオーバライドするように，独自の派生クラスを定義すればよい。

```
class ClientButton : public Button {
    // ...
    void action();
    // ...
};

void ClientButton::action() {
    // この型のボタンが押された
    // ときに実行するためのコード
    // ...
}
```

ここで，ClientButtonは調整済みのactionを与えている。これで，アプリケーションは，この派生クラスを使って，調整済みボタンを作成することができる。

```
Button *bp = new ClientButton;
add_button_to_gui( bp );
```

コールバック機構として仮想関数のオーバライド機能を使うには，利用者コードのプログラマは，派生クラスを正しく定義し，仮想関数をオーバライドできる程度まで，ライブラリのクラスを理解している必要がある。これをプログラマが楽に行うためには，ライブラリインターフェイスの抽象デザインはわかりやすく，かつ，インターフェイスプロトコルは明確に定義され，ドキュメントが存在しているべきである。また，継承機構では，実行時にコールバックの動作を再設定することができない。以下に，コールバックを格納するのに，関数へのポインタ型のデータメンバを使って，ボタンを実装する例を示す。

```
typedef void (*Action_t)();
class SettableButton : public Widget {
    // ...
    Action_t callback;
    void action() { if( callback ) callback(); }
    void set_action( Action_t afp ) { callback = afp; }
    // ...
};
```

この場合，派生クラスを定義する代わりに，set_actionを呼び出して，コールバック関数を登録する。ここで，コールバックが設定されていない場合には，何らかのデフォルトの動作を実行するようにaction関数を実装することもできるだろう。

8.4　拡張性のあるライブラリ

ソフトウェアライブラリは，多くのプログラムで使われることを目的に作成されているが，実際の問題として，プログラムが潜在的に必要とするすべての機能をライブラリの実装に与えることは不可能に近い。多くの場合，ライブラリは，アプリケーションが必要とするものに近い機能を与えるが，厳密に要求されているものを与えるわけではない。こうした場合，ライブラリは，もとのライブラリのデザインと実装に手を入れることなく新しいアプリケーションが必要とする機能を追加できるように，拡張性を考えてデザインされていると便利である。ライブラリデザインの機能を拡張し，変種を作成する汎用機構として継承を使うこともできる。

拡張性を考えて，継承の階層構造をデザインする際の 1 つの問題は，いかに適切に抽象概念を層構造として表現するかということにある。第 5 章の抽象構文木節の階層を再び考えてみよう。このライブラリは，二項演算子の汎用の抽象概念を与える。継承を使ってライブラリを拡張することで，扱うことのできる二項演算子の種類を容易に増やすことができる。

```
class And : public Binop {
  public:
   And( Node *l, Node *r ) : Binop(l,r) {}
   int eval() const
      { return left->eval() & right->eval(); }
};
```

新しい二項演算子を作成するには，既に存在する汎用の抽象概念に，追加しようとする演算子の仕様を追加するだけである。

しかしこのライブラリには，単項マイナス演算子は定義されているが，単項演算子の汎用の抽象概念は定義されていない。このため，新しい単項演算子を追加するには，すこしばかり仕事が必要であり，追加しようとする単項演算子の特徴のほかに，単項演算子の汎用の特性(たとえば，コンストラクタとデストラクタ)を定義しなければならない。

```
class Uplus : public Node {
    Node *operand;
  public:
```

```
    Uplus( Node *o ) { operand = o; }
    ~Uplus() { delete operand; }
    int eval() const { return operand->eval(); }
};
```

ライブラリに単項演算子の汎用の抽象概念が与えられていれば，このクラスUplusの定義のほとんどの部分は不要であろう。こうした抽象概念が不在のため，ライブラリの利用者は，この新しいクラスを追加するために，本来は継承されるはずのコードを作成するほかに，ライブラリの実装の細部まで詳しく理解する必要がある。しかし，拡張機能を実装しようとするプログラマが，ライブラリの実装の細部すべてを正確に理解することは困難だろう。一般に，ライブラリがどのように使われ，どのように拡張されるかをあらかじめ知ることはできないので，常にあらゆるレベルの抽象概念を意識して実装を作成すべきである。

継承の階層構造をデザインする際のもう1つの問題は，クラスインターフェイスの中で，調整あるいは置換してもよい汎用の演算をどのようにして示すかということである。これらの演算は，ライブラリの基礎実装から変種を作成するために派生クラスを定義できるように，仮想関数として宣言されるべきである。

ライブラリは，汎用の抽象概念にはデフォルトの動作を与えずに，抽象概念の各変種にだけ独自機能を与えてもよい。この場合，基底クラスの仮想関数の実装に適したものは存在しないことになる。実際に，特定の変種を除けば，こういう抽象概念のインスタンスを実際のオブジェクトとして生成することは誤りだと思われる。この種の抽象概念を明示的に表現するには**抽象クラス**(abstract class)が使われる。

ここで，第6章のDevice(デバイス)クラスを再び取り上げよう。すべてのデバイスは同じインターフェイスを持ち，このインターフェイスはDevice基底クラスから継承されるべきである。各種デバイスは，read，write，ioctlに独自の機能を与える必要があり，これらの演算にはデフォルトの動作はない。以下のDeviceクラスは，仮想関数を純粋仮想仮数とすることによって，自分の派生クラスに仮想関数の独自バージョンを定義することを強制している。純粋仮想関数を持つクラスは抽象クラスであり，このクラスのオブジェクトを作成することはできない。さらにDeviceクラスは，コンストラクタをprotectedにすることによっても，派生クラスでのみ使われるようにデザインされている。

```
class Device {
  protected:
   Device();
  public:
   virtual ~Device();
   virtual int open( char *, int, int );
   virtual int close( int );
   virtual int read( int, char *, unsigned ) = 0;
   virtual int write( int, char *, unsigned ) = 0;
   virtual int ioctl( int, int, ... ) = 0;
};
```

Deviceからの派生クラスが，これらの純粋仮想関数を独自にオーバライドしなかった場合，派生クラス自体も抽象クラスであり，オブジェクトとしてインスタンスを生成することはできない。

```
class Disk : public Device {
  public:
   Disk();
   ~Disk();
   int open( char *, int, int );
   int close( int );
   int read( int, char *, unsigned );
   int write( int, char *, unsigned );
   // 誤り: ioctlを忘れている
};
//...
Device *dp = new Disk;  // エラー!
```

抽象クラスは，ライブラリにおけるオブジェクトインターフェイスの必須プロトコルを明示的に与えるために使うことができる。インターフェイスプロトコルを，ライブラリの実装のほかの部分から分離すれば，ライブラリの機能を拡張するために新しいクラスを追加する際に，何が要求されているかを明らかにすることができる。

継承が，拡張性のあるライブラリを作成するためのただ1つの手段というわけではない。いくつかのライブラリは，拡張性のために細かい演算群と構文上の枠組みを与えている。C++標準のストリーム入出力ライブラリiostreamは，利用者定義の型のために拡張でき，型の検査を行う入出力機能を与える。このライブラリは，新しい型に対して入出力演算を多重定義することで拡張を行う。これと対称的に，Cの標準入出力ライブラリstdioの入出力機能は，型検査は行わず，拡張も簡単ではない。

```
#include <stdio.h>

extern int age;
extern String name;
printf( "%s's age is %d\n", age, name ); // oops!
```

これは「正しい」C++のコードであるが，実行すると意味のない結果を表示し，運が悪いとプログラムを破壊する可能性さえある。書式文字列に対応する引数ageとnameの順序が間違っているにもかかわらず，printfの書式文字列が“char *”，intという順序の引数を期待しているという型情報をコンパイラに伝えられないため，この誤りは検出されない[1]。また，たとえ引数を正しい順序にしても，まだ期待通りには表示されない。printf関数は，書式文字列の%sに対して“char *”型の引数を期待しているが，与えられるのはString型の引数である。しかし，C++コンパイラには“char *”型が必要であるという型情報が与えられないので，“String::operator const char *”によるStringから“char *”へ暗黙の型変換は行われない。

```
inline int
age_print( char *n, int a )
   { return printf( "%s's age is %d\n", n, a ); }
//...
age_print( name, age );
```

このようにして必要な型情報を提供することもできないわけではないが，このアプローチは，すべての表示演算に対して特別の処理を行うことを強制してしまう。

ライブラリiostreamは，より安全なアプローチを採用している。型検査付きの入出力演算のために，iostreamでは，演算子“>>”と“<<”を多重定義していることに注目してほしい。先程の表示命令は以下のように書くことができる。

```
cout << name << "'s age is " << age << "\n";
```

以下に，ライブラリiostreamのいくつかの機能を省略したバージョンを示す。最初は，左シフト演算子(出力ストリームに値を挿入する「挿入演算子」)が入った出力ストリームクラスの定義である。

---

[1] 訳注：最近のGCC(GNU Cコンパイラ)は，書式文字列の型情報から引数の型検査を行う。

```
typedef FILE *FP;

class ostream {
   FP f;
  public:
   ostream( FP fl ) : f( fl ) {}
   operator FP() { return f; }
   ostream &operator <<( const char * );
   ostream &operator <<( int );
   ostream &operator <<( long );
   ostream &operator <<( double );
};

ostream cout = stdout;
```

この簡略バージョンでは，シフト演算子を多重定義する際にライブラリstdioの機能を使用する。

```
ostream &
ostream::operator <<( const char *s ) {
   fprintf( f, "%s", s );
   return *this;
}
```

ほかの型(intやlong)に対する“ostream::operator <<”の定義も同様である。ここでシフト演算子が実行の対象となったostreamオブジェクト(へのリファレンス)を返していることに注目してほしい。これにより，複数の出力演算をただ1つの(コンマなしの)式に続けて書くことができる。シフト演算子は，C++のほとんどの演算子と同様に左から右に評価されるので，次の式

```
cout << name << "'s age is " << age << "\n";
```

は，

```
(((cout << name) << "'s age is ") << age) << "\n";
```

と等価である。この式の各“<<”演算子は，クラスostreamのオブジェクトcoutを返すため，これは，

```
cout << name;
cout << "'s age is ";
cout << age;
cout << "\n";
```

と等価である。しかし，前者のほうがはるかに読みやすい。

入力ストリームも同様にして作成され，右シフト演算子(入力ストリームから値を取り出す「抽出演算子」)が定義される。

```
class istream {
   FP f;
  public:
   istream( FP fl ) : f(fl) {}
   operator FP() { return f; }
   istream &operator >>( char * );
   istream &operator >>( int & );
   istream &operator >>( long & );
   istream &operator >>( double & );
   int gchar();    // 1文字を得る
};

istream &
istream::operator >>( long &l ) {
   fscanf( f, "%ld", &l );
   return *this;
}

istream cin = stdin;

cout << "Your age and name: ";
int age;
char nm[32];
cin >> age >> nm;
```

クラスostreamとistreamは,「拡張性のある」入出力ライブラリの基礎となるC++の組み込み型に対する入出力演算と，入出力演算のための(一例としての)構文を提供している。利用者定義の型への拡張は，この入出力ストリームの枠組みに，シフト演算子の多重定義を追加することで行う。たとえば，このストリーム入出力iostreamで複素数を扱う場合を考えてみる。

```
class complex {
   double re, im;
  public:
```

```
    friend ostream &
    operator <<( ostream &, const complex & );
    friend istream &
    operator >>( istream &, complex & );
    // ...
};
```

このフレンド宣言は，シフト演算子に複素数のprivate部をアクセスさせるために必要である。複素数入出力のためのシフト演算子は，ちょうどcomplex型が組み込みの型(double)を利用して定義されたように，組み込みのシフト演算子を利用して以下のように定義される。

```
ostream &
operator <<( ostream &out, const complex &c ) {
    out << "(" << c.re << "," << c.im << ")";
    return out;
}

istream &
operator >>( istream &in, complex &c ) {
    in >> ws;    // 演習8-7参照
    if( in.gchar() != '(' ) error();
    in >> c.re;
    in >> ws;
    if( in.gchar() != ',' ) error();
    in >> c.im;
    in >> ws;
    if( in.gchar() != ')' ) error();
    return in;
}
```

出力演算子“<<”は，入力演算子“>>”が解釈できるフォーマットで複素数を出力を行うことに注意されたい。

もっと複雑な利用者定義の型(たとえば，抽象構文木など)の出力も可能である。たとえば，ポインタnpがある構文木の根を指している際に，以下のコードで，構文木の内容を括弧で囲まれた中置形式の集まりとして表示することを考えてみよう。

```
extern Node *np;
cout << "The expression is: " << np;
```

まず，(仮想関数に責任を分担した)表示能力を，抽象構文木の節の階層に組み込ん

でいくことから始める。

```
class Node {
   // ...
   virtual void print( ostream & ) const = 0;
};

class Binop : public Node {
   // ...
   virtual void print_op( ostream & ) const = 0;
   void print( ostream &out ) const {
      out << "(";
      left->print( out );
      print_op( out );
      right->print( out );
      out << ")";
   }
};

class Plus : public Binop {
   // ...
   void print_op( ostream &out ) const { out << "+"; }
};

class Int : public Node {
   // ...
   void print( ostream &out ) const { out << value; }
};

// etc. ...
```

仮想関数printの動作は，この節の階層に既に用意されている仮想関数evalや仮想デストラクタの動作と類似している。つまり，表示動作では，対象とする部分木の根の型に応じて，適切なprint関数を再帰的に呼び出しながら，抽象構文木の内容を表示していくことになる。

あとは，既存のライブラリiostreamにメンバ関数printに関する記述を組み入れるだけである。

```
ostream &
operator <<( ostream &out, const Node *np ) {
   np->print( out );
   return out;
}
```

ところで，この1つ前の例に，利用者にメッセージを表示して入力を要求して返答を待つという処理があった。

```
cout << "Your age and name: ";
```

残念ながら，これは意図した通りの動作をしないかもしれない。出力ストリームcoutがバッファリングを行う場合，利用者へのメッセージが実際に表示されるのは，(期限ははっきりとしないが)出力バッファが満たされるまで延期されてしまう。正しく動作させるためには，coutのバッファをフラッシュし，メッセージが実際に表示される時期を確実にしておくべきである。

```
cout << "Your age and name: ";
flush( cout );
```

しかし，こういった制御信号を入出力演算のあちこちに埋め込んでしまうと，逆にコードの可読性を低下させ，プログラムの作成者と読者の両者にプログラムの論理を取り違えさせてしまう。これはむしろ，

```
cout << "Your age and name: " << flush;
```

と書いたほうがよいだろう。

こういった処理を可能にするものを**適用子**(applicator)と**操作子**(manipulator)と呼ぶ。操作子とは，式中で使われる値で，式の第1の目的に厳密には無関係の副作用を起こすもののことである。概して，操作子の値は関数のアドレスである。

```
ostream &flush( ostream & );
```

関数アドレスを出力式で扱えるようにするため，引数で指定された関数(ポインタ)を出力ストリームに適用し，出力ストリームを返す適用子を定義する。

```
typedef ostream &(*Manip)(ostream &);
inline ostream &
operator <<( ostream &out, Manip f )
    { return f( out ); }
```

この適用子により，Manip型の操作子なら何でもシフト演算子の出力式で扱えるようになる。以下のように，ほかの型の操作子のために，適用子の定義を追加していくこともできる。

```
typedef int (*Manip2)(FILE *);

inline ostream &
operator <<( ostream &out, Manip2 f ) {
   f( out );
   return out;
}

extern int close( FILE * );

cout << "So long..." << flush << close;
```

操作子がパラメータを持てると，いろいろと便利である。たとえば，出力ストリームの各種のデータ伝送特性を，操作子の引数で設定できるようにしてみよう。

```
enum { odd, even, none };
enum { slow = 1200, lagom = 4800, fast = 9600 };

cout<<speed(fast)<<parity(none)<<"login: "<<flush;
```

この場合の操作子の値は，関数アドレスと引数の整数値の両方から成るクラスオブジェクトである。

```
struct act_rec {
   void (*fp)(FILE *, int);
   int arg;
   act_rec( void (*f)(FILE *, int), int a )
      : fp(f), arg(a) {}
};
```

また，speedやparityは，以下のact_recを返す通常の関数である。

```
inline act_rec
speed( int baudr ) {
   extern void set_speed( FILE *, int );
   return act_rec( set_speed, baudr );
}
```

最後に，操作子act_recを出力ストリームostreamに適用させる適用子を定義すれば，パラメータを持つ操作子の実装が終了する。

```
inline ostream &
operator <<( ostream &out, act_rec m ) {
   m.fp( out, m.arg );
   return out;
}
```

演　習

演習 8-1† この章のostreamのように，段階的にメッセージを構築することができるメッセージバッファ型をデザインし，文字列，数値型，文字，ほかのメッセージバッファを挿入する演算子と，構築された文字列を抽出する手段を与えよ。デザインにあたっては，次のことに注意されたい。メッセージバッファ型には，挿入演算のほかに，どういった演算があると便利か。構築中のメッセージを格納するバッファのメモリは誰が確保し，管理するのか(メッセージバッファ型あるいは利用者)。メモリはいつ解放されるのか。どういったエラー状態が起りうるか，エラーの情報はどのように利用者に通知すべきか。

演習 8-2† 演習 8-1 のメッセージバッファ型の実装を修正して，利用者がバッファがあふれたときに別様の動作をする派生クラス型を定義できるようにせよ

演習 8-3 演習 8-1 と 8-2 のメッセージバッファ型のメモリ管理の実装を修正して，利用者がこの型のオブジェクトをフリーストア上に割り当てることができないようにせよ。

演習 8-4 各自の環境で与えられる時間を扱う機能を調べてみよ。単純かつ使いやすいか。多重スレッド環境でも安全に使うことができるか。エラー状態は明らかに指定されているか，容易に検出できるか。また，この時間機能を使うコードはほかの環境に容易に移植できるか。これらの条件を満たすように，時間機能をまとめた型をクラスを使ってデザインせよ。

演習 8-5 C++を，ベクトルと行列の算術演算のアプリケーション指向言語にするためのライブラリを開発せよ。

演習 8-6 関係データベース代数のためのライブラリをデザインせよ。データベース演算のインターフェイスとして，中置形式の演算子のものと関数呼び出し構文のもの2種類を実装せよ。**Hint:** 抽象構文木の探索といった特性をどのように表現すればよいか。データベースアクセスソフトウェアとプログラミング言語インタープリタの違いは何か。リレーショナル代数をリレーショナル計算法に拡張するための制御構造をデザインせよ。

演習 8-7 入力ストリームistreamから空白類を取り除く操作子と適用子を書け。

```
cin >> ws;
```

出力ストリームostreamに改行を挿入し，出力バッファをフラッシュする操作子を書け。

```
cout << endl;
```

演習 8-8† クラスcomplexの加算演算で，結果をまるめる(四捨五入する)操作子と適用子を書け。

```
extern complex a, b, c, d;
d = a + b + round + c + round;
```

同様の演算を(組み込みの)整数の式で行う操作子を書け。

演習 8-9 型“void (*)(char *)”の操作子を型“char *”の引数に適用する適用子を書け。

演習 8-10 第4章のStringクラスを修正して，String式の中で型“void (*)(String &)”と型“void (*)(char *)”の操作子を適用できるようにせよ。

演習 8-11† 第4章のSortedCollection型に対する挿入演算子(<<)を書け。この際に，SortedCollectionの実装とインターフェイスを変更してはいけない。

演習 8-12 この章で述べた抽象構文木の挿入演算子に対応する抽出演算子を書け。ただし，入力全体が括弧で囲まれていると仮定することはできない(**Hint:** 第5章の電卓言語の構文解析系とこの抽出子の違いは何か)。

演習 8-13　この章の`iostream`の定義が与えられている際に，“`cfile >> first << replacement`”という式がコンパイルできない理由を説明せよ。こういった混合の入出力式が許されるように，`iostream`の実装を再度デザインせよ。

演習 8-14　ライブラリ`iostream`の挿入演算子と抽出演算子として，“<<” や “>>” は優れた選択だろうか，悪い選択だろうか。ほかの演算子としては，どれが優れた選択だろうか。どうして “->” あるいは “+=” は悪い選択だろうか。

演習 8-15　クラス型の調整に継承を使って，ライブラリのソースを変更することなくライブラリのバグを修正する手法を示せ。このテクニックを，演習 5-4 の解答で指摘した，抽象構文木の`eval`関数が二項演算子節の引数を受け取った場合，評価の順序は環境に依存してしまう問題を解消するために使用せよ。

演習 8-16†　ライブラリは，使用にあたって最初に初期設定を要求することが多い。どのようなアプリケーションと組み合わせて使っても，ライブラリを(1 回だけ)初期設定することを保証する機構をデザインせよ。

演習 8-17　いくつかの調整可能なライブラリは「アプリケーションの枠組み」としてデザインされ，実際のアプリケーション群を作成する際に，白紙のアプリケーションとして使われる。

演習 6-5 を参考にして，連続する状態遷移として単位分解されるアプリケーションのための枠組みをデザインせよ。この枠組みを調整するのに継承を使い，航空予約システム，C++ の語彙解析，演習 4-17 の有限状態オートマトン抽象データ型を実装せよ。

演習 8-18　ISAM データベースパッケージの低レベルインターフェイスを考えよう。データベースに関する概念としては，ファイル，インデックス，キー，プロパティ，シーケンシャルアクセス，ランダムアクセスがある。低レベルインターフェイスは，これらの概念をどのように表現すればよいか。このインターフェイスを修得したり，使ったりするのはどの程度容易か，また，偶然に誤って使われることがどの程度あるだろうか。これらの概念を明確にし，エラーの起りやすさを減らした ISAM パッケージの C++ インターフェイスをデザインせよ。

演習 8-19　演習 8-18 では，もともとのインターフェイスのすべての様相を各自の C++ バージョンに表現することができただろうか。また，もともとのインターフェイスの様

相のうち，各自の概念上の枠組みにあてはめるにはあまりに低レベルすぎるものがあるだろうか。必要であれば，もともとのインターフェイスの低レベル機能へのアクセスを許しつつ，まったくの概念上の枠組み管理する機構をデザインせよ。

演習 8-20 2.6 節の関数テンプレート`input`を考えよう。このテンプレートは，`complex`にインスタンスを生成することができるだろうか。この問題に対して，ストリーム入出力はどのように役に立つか。

A

演習の解答

演習 1-2

```
a        // int
b        // int *
*b       // int
c        // int *
*c       // int
d[2]     // int *
*d       // int *
**d      // int
c[-2]    // int
c-2      // int *
*(c-2)   // int
&c       // int **
```

□

演習 1-3

```
c = cc;           // 正しい
cc = c;           // 誤り
pcc = &c;         // 正しい
pcc = &cc;        // 正しい
pc = &c;          // 正しい
pc = &cc;         // 誤り
pc = pcc;         // 誤り
pc = cpc;         // 正しい
pc = cpcc;        // 誤り
cpc = pc;         // 誤り
*cpc = *pc;       // 正しい
pc = *pcpc;       // 正しい
**pcpc = *pc;     // 正しい
*pc = **pcpc;     // 正しい
```

□

演習 2-4

```
char *
get_token( char *&s, char *ws = " \t\n" ) {
   char *p;
   do
      for( p = ws; *p && *s != *p; p++ );
   while( *p ? *s++ : 0 );
   char *ret = s;
   do
      for( p = ws; *p && *s != *p; p++ );
```

```
    while( *p ? 0 : *s ? s++ : 0 );
    return ret;
}

int strlen( char *s ) {
    char *t = get_token( s, "" );
    return s-t;
}
```

□

演習 3-1 double 型引数のビット表現を得るために共用体を使用する。

```
void
bitprint( double val ) {
    static int nbitsinchar = 0;
    if( !nbitsinchar ) {
        // 型 charの占めるビット数を計算する。
        for( unsigned char c = 1; c; c <<= 1 )
            nbitsinchar++;
    }
    union {
        double d;
        unsigned char a[ sizeof(double) ];
    };
    d = val;
    for( int i = 0; i < sizeof(double); i++ ) {
        unsigned c = a[i];
        for( int j = nbitsinchar; j; j-- )
            if( c & ( 1 << j ) )
                printf("1");
            else
                printf("0");
    }
}
```

□

演習 3-2 抽象リスト型を実装するため，リング状リストを使う。

```
class PtrList {
  public:
   PtrList()
      : last( 0 ) {}
   ~PtrList();
   void prepend( void * );
   void append( void * );
```

```
      void *pop_front();
      int empty() const;
    private:
      struct El {
         void *p;
         El *next;
      } *last; // リング状
};

void
PtrList::prepend( void *v ) {
   El *el = new El;
   el->p = v;
   if( last ) {
      el->next = last->next;
      last->next = el;
   }
   else
      last = el->next = el;
}

void
PtrList::append( void *v ) {
   prepend( v );
   last = last->next;
}

void *
PtrList::pop_front() {
   if( last ) {
      El *tmp = last->next;
      if( last == tmp )
         last = 0;
      else
         last->next = tmp->next;
      void *ret = tmp->p;
      delete tmp;
      return ret;
   }
   else
      return 0;
}

int
PtrList::empty() const
   { return last == 0; }
```

□

演習 3-3 ハッシュ表クラスHtabは，同じ場所へ「ハッシュ」されたレコードをリスト構造で格納することにすれば，以下のように実装できる。

```
class Rec {
   // 実装の細部...
  public:
   const char *key() const;
};

class Htab {       // ハッシュ表型
  public:
   Rec *insert( Rec * );
   Rec *lookup( const char *key ) const;
   Rec *remove( const char *key );
   Htab(int size = 64 );
   ~Htab();
  private:
   int size;         // 表のサイズ
   class Clist {  // ハッシュ表による衝突リスト
      Rec *r;
      Clist *next;
      friend class Htab;
   } **tab;          // 表へのポインタ
   static int hash( const char * );
};

Htab::Htab( int sz )
   : size( sz ), tab( new Clist *[size] ) {}

Htab::~Htab() {
   for( int i = 0; i < size; i++ ) {
      Clist *p = tab[i], *q;
      while( p ) {
         q = p;
         p = p->next;
         delete q;
      }
   }
   delete [] tab;
}
Rec *
Htab::lookup(  const char *s ) const {
   // レコードへのポインタを返す。失敗のときは 0 を返す。
```

```
    for( Clist *p = tab[hash(s)]; p; p = p->next )
        if( strcmp( s, p->r->key() ) == 0 )
            return p->r;
    return 0;
}

Rec *
Htab::insert( Rec *r ) {
    // 挿入したレコードへのポインタを返す。失敗のときは 0 を返す。
    if( lookup( r->key() ) )
        return 0;
    int i = hash( r->key() );
    Clist *p = new Clist;
    p->r = r;
    p->next = tab[i];
    tab[i] = p;
    return r;
}

Rec *
Htab::remove( const char *s ) {
    // 削除したレコードへのポインタを返す。失敗のときは 0 を返す。
    int i = hash( s );
    Clist *p = tab[i], *q = 0;
    while( p && strcmp( p->r->key(), s ) ) {
        q = p;
        p = p->next;
    }
    if( p ) {
        if( q )
            q->next = p->next;
        else
            tab[i] = p->next;
        Rec *r = p->r;
        delete p;
        return r;
    }
    else
        return 0;
}
```

Htab::hash には，1.3 節のハッシュ関数hashpjw が利用できる。□

演習 3-8　最初に，自分が宣言されている関数あるいはブロックの名前で初期設定されるクラス型Trace を定義する。

```
class Trace {
   char *msg;
  public:
   Trace( const char *m )
      : msg( strcpy( new char[strlen(m)+1], m ) )
      { fprintf( stderr, "entering %s\n", msg ); }
   ~Trace() {
      fprintf( stderr, "leaving %s\n", msg );
      delete [] msg;
   }
};
```

このクラスTraceのオブジェクトを宣言すれば，オブジェクトがスコープに入るときに，コンストラクタが“entering …”というメッセージを表示し，オブジェクトがスコープから出るときに，デストラクタが“leaving　…”というメッセージを表示する。以下のコードで，Traceオブジェクトがどのように動作するか説明せよ。

```
Trace x = "global environment";

void
buggy_func( int arg ) {
   Trace x = "buggy_func";
   if( arg ) {
      Trace x = "then part";
      //...
      return;
   }
   else {
      Trace x = "else part";
      while( arg-- ) {
         Trace x = "while loop";
         if( arg > 6 )
            return;
      }
      //...
   }
}
```

たとえば，Traceオブジェクトが作成されたときに資源としてのメッセージテキストが確保され，オブジェクトが破壊されたときに解放されると考えれば，このテクニックは，プログラミングデザインでのほかの資源管理にどのように適用することができるだろうか。□

演習 3-14

```
int A::*a;
```

は，Aの整数メンバへのポインタ。

```
int A::*b();
```

は，Aの整数メンバへのポインタを返す関数。

```
int *A::*c;
```

は，Aの整数メンバへのポインタへのポインタ。

```
int *A::* const d = 0;
```

は，Aのメンバ(整数へのポインタ)への定数ポインタ。

```
int const * A::* const e = 0;
```

は，Aのメンバ(定数整数へのポインタ)への定数ポインタ。

```
int A::*B::*f;
```

は，Bのメンバ(Aの整数メンバへのポインタ)へのポインタ。

```
int A::**B::*C::**const&g = 0;
```

は，Cのメンバ(Bのメンバ(Aの整数メンバへのポインタへのポインタ)へのポインタ)へのポインタへの定数ポインタへのリファレンス。

```
int A::*(A::*h)(int A::*);
```

は，引数としてAの整数メンバへのポインタを取り，Aの整数メンバへのポインタを返す，Aのメンバ関数へのポインタ。

```
int (A::*i);
```

は，Aの整数メンバへのポインタ。

```
int (A::*j)();
```

は，引数を取らず，整数を返すAのメンバ関数へのポインタ。□

演習 4-5　抽象データ型スタックは演算によって定義される。ここではpush，popおよび状態関数(empty，full)を与えることにする。また，初期設定と破壊が適切に行われるように，コンストラクタとデストラクタも与えている。

```
typedef int T;

class Stack {
  public:
   Stack( size_t = 32 );
   ~Stack();
   int empty() const;
   int full() const;
   void push( T );
   T pop();
  private:
   const int max;
   int i;
   T *s;
   Stack( Stack & );
   void operator =( Stack & );
};
```

スタックのコンストラクタは，スタックというデータ構造の役目を果す配列を確保し，現在のインデックスと最大のインデックスを設定する。

```
Stack::Stack( size_t depth )
   : max( depth ), i( 0 ),
   s( new T[ depth ] ) {}
```

スタックのほかの演算は単純であるから，インライン関数にしてもよい。

```
inline
Stack::~Stack()
   { delete [] s; }

inline int
Stack::empty() const
   { return i == 0; }

inline int
Stack::full() const
   { return i >= max; }
```

```
inline void
Stack::push( T el ) {
   if( !full() )
      s[ i++ ] = el;
}

inline T
Stack::pop()
   { return s[ empty() ? i : --i ]; }
```

固定サイズの配列の代わりに連結リストを使った実装に変更することを考えよう。新しい実装に応じて，スタック型のインターフェイスと動作を変更すべきだろうか。□

演習 4-10

```
struct El {
   void *p;
   El *next;
};

template <class T>
class PtrList {
  public:
   PtrList()
      : last( 0 ) {}
   ~PtrList();
   void prepend( T * );
   void append( T * );
   T *pop_front();
   int empty() const;
  private:
   El *last;    // リング状
};

template <class T>
void
PtrList<T>::prepend( T *v ) {
   El *el = new El;
   el->p = v;
   if( last ) {
      el->next = last->next;
      last->next = el;
   }
   else
      last = el->next = el;
}
```

```
template <class T>
void
PtrList<T>::append( T *v ) {
   prepend( v );
   last = last->next;
}

template <class T>
T *
PtrList<T>::pop_front() {
   if( last ) {
      El *tmp = last->next;
      if( last == tmp )
         last = 0;
      else
         last->next = tmp->next;
      T *ret = (T *)tmp->p;
      delete tmp;
      return ret;
   }
   else
      return 0;
}
template <class T>
int
PtrList<T>::empty() const
   { return last == 0; }
```

□

演習 4-12 ここでは，型変換を行う際に暗黙のうちに適用されてもよいのは，1つの利用者定義の型変換(単一引数のコンストラクタか型変換演算子)だけであるという事実を利用するアプローチを使う。

```
struct DateInit {
  public:
   DateInit( const char *s )
      : date( s ) {}
  private:
   const char *date;
   friend class Date;
};
```

```
class Date {
   // ...
   Date( DateInit &init );
   // ...
};
```

文字列によるDateの初期設定は，利用者定義の型変換1つだけを必要とするので，依然として成功するが，条件式の引数型変換では，2つの利用者定義の型変換を必要とするであろう。

```
Date d( "tomorrow" );    // OK: DateInit( "tomorrow" )
if( d > "today" )        // エラー！ Date(DateInit("today"))
```

□

演習4-14 “operator <”の実装では，strcmpを呼び出す前に，String引数を文字列に型変換する型変換演算子が使われる。

```
int
operator <( const String &a, const String &b )
   { return strcmp( a, b ) < 0; }
```

効率のいい“operator +”を実装するために，StringとString::Repにprivateコンストラクタを追加した。これによって，Stringあるいは“const char *”による初期設定に既存のpublicコンストラクタを使う代わりに，関数の戻り値を“String::Rep *”で直接初期設定することができる。

```
String
operator +( const String &a, const String &b )
   { return new String::Rep( a, b ); }

// 新規にクラスStringに追加されたprivateコンストラクタ
String::String( Rep *rep )
   : r( rep ) {}

// 新規にクラスString::Repに追加されたprivateコンストラクタ
String::Rep::Rep( const char *a, const char *b )
   : str(strcat(strcpy(new char[strlen(a)+strlen(b)+1],
      a),b)) {}
```

“operator +=”の実装は，“operator +”と似ているが，同時に“operator =”の実装のように代入の意味も持つ必要がある。

```
String &
String::operator +=( const String &s ) {
   const char *a = r->str;
   String::Rep *rtmp = r;
   r = new String::Rep( a, s );
   if( !--rtmp->refs )
      delete rtmp;
   return *this;
}
```

□

演習 4-18　制御抽象型をSortedCollectionのフレンドとして宣言する。この制御抽象型には，反復を初期状態に戻したり，次の要素に移動したり，引数の述語論理を満たす次の要素に移動したりという演算を与える。

```
typedef int (*Pred)( T );

class Ctl {
  public:
   Ctl( const SortedCollection &sc )
      : c( sc ), i( 0 ) {}
   ~Ctl()
      {}
   void reset()
      { i = 0; }
   int done() const
      { return i >= c.free; }
   void next()
      { i++; }
   void next( Pred );
   T get() const
      { return c.ary[i]; }
  private:
   const SortedCollection &c;
   int i;
};

void
Ctl::next( Pred p ) {
   while( ++i < c.free )
      if( p( c.ary[i] ) )
         break;
}
```

Ctlの使用法は単純である。

```
int odd( int i ) { return i & 1; }
//...
for( Ctl ctl = c; !ctl.done(); ctl.next() )
   // ...
ctl.reset()
ctl.next( odd );
// ...
```

□

演習 4-20

```
typedef void (*Manip)( T & );

class Tree {
  public:
   Tree()
      : root( 0 ) {}
   ~Tree()
      { delete root; }
   int insert( T );
   void inorder_apply( Manip m )
      { if( root ) root->apply( m ); }
  private:
   class Node {
     public:
      Node *left, *right;
      T el;
      Node( T e )
         : left( 0 ), right( 0 ), el( e ) {}
      ~Node();
      int insert( T );
      void apply( Manip );
   } *root;
   friend class Inorder;    // 次の演習のために
};

int
Tree::insert( T el ) {
   if( root )
      return root->insert( el );
   root = new Node( el );
   return 1;
}

int
Tree::Node::insert( T arg ) {
   if( arg == el )
```

```
            return 0;
        if( arg < el )
            if( left )
                return left->insert( arg );
            else {
                left = new Node( arg );
                return 1;
            }
        if( right )
            return right->insert( arg );
        right = new Node( arg );
        return 1;
    }

    void
    Tree::Node::apply( Manip m ) {
        if( left )
            left->apply( m );
        m( el );
        if( right )
            right->apply( m );
    }
```

□

演習 4-21　通りがけ順の制御抽象は，本質的には演習 4-20 の通りがけ順の`apply`関数のように動作する。しかし，`apply`関数はすべてのノードに関数を適用して回るが，制御抽象は，各ノードごとに必ず復帰し，次は前に走査を中断したところから始めるところが違う。ここでは「関数のアクティベーション(活動)情報」のスタックを管理することで，再帰の通りがけ順関数をシミュレートする制御抽象を作成することにする。

```
    class Inorder {
      public:
        Inorder( const Tree &t )
            : top( t.root ? new ActRec( t.root, 0 ) : 0 )
            { next(); }
        ~Inorder();
        int done() const
            { return top == 0; }
        void next();
        T get() const
            { return top->root->el; }
      private:
        class ActRec {          // アクティベーションレコードをシミュレートする
          public:
```

```
        Tree::Node *root; // 局所データ
        enum { LEFT, VISIT, RIGHT, RETURN } pc;
        ActRec *prev;     // 前の呼び出し
        ActRec( Tree::Node *r, ActRec *p )
           : root( r ), pc( LEFT ), prev( p ) {}
   } *top;                  // アクティベーションスタックの先頭
};
```

next 関数は，プログラムカウンタに応じて分岐し，アクティベーションレコードをスタックに正しく入れたり，取り出したりする。このアルゴリズムは，Tree::Node::apply で使われているものと同じである。

```
void
Inorder::next() {
   while( top )
      switch( top->pc ) {
      case ActRec::LEFT:
         top->pc = ActRec::VISIT;
         if( top->root->left )
            top = new ActRec( top->root->left, top );
         break;
      case ActRec::VISIT:
         top->pc = ActRec::RIGHT;
         return;
      case ActRec::RIGHT:
         top->pc = ActRec::RETURN;
         if( top->root->right )
            top = new ActRec( top->root->right, top );
         break;
      case ActRec::RETURN:
         ActRec *tmp = top;
         top = top->prev;
         delete tmp;
         break;
      }
}
```

□

演習 4-26　連想配列は，インデックスの値を配列の要素に対応付けるものである。この定義に従えば，C++の標準の配列型は，整数を配列要素に対応付ける連想配列である。コンパイラの記号表も，識別子と属性を対応付ける連想配列の 1 つの例である。ここで考える連想配列は，String を String に対応付けるもので，String はインデックスの型でもあり，要素の型でもある。

```
typedef String I;
typedef String E;
```

StringからStringへの対応付けを実装するために，連想配列クラスを作成し，あたかも通常の配列であるように連想配列にインデックスを指示するために演算子[]を多重定義する。

```
class Pair {
   I index;
   E el;
   Pair *next;
   friend class Aary;
};

class Aary {
  public:
   Aary()
      { elems = 0; }
   ~Aary();
   E &operator []( I );
  private:
   Pair *elems;
   Aary( Aary & );
   void operator =( Aary & );
};
```

この実装では，配列はインデックスと要素の組みのリストとして構成される。ここで，“operator []”は，連想配列へのインデックス付けのたびに線形探索を行うことに注意しよう。これは配列のインデックス数が少なければ適当であるが，大きい配列にはさらに効率のよい探索法(ハッシュ表，平衡木など)が必要である。

```
E &
Aary::operator []( I i ) {
   for( Pair *p = elems; p; p = p->next )
      if( p->index == i )
         return p->el;
   p = new Pair;
   p->index = i;
   p->next = elems;
   elems = p;
   return p->el;
}
```

“operator []”は，アクセス時に目的の要素が不在であれば，新たに要素を作成

することに注意してほしい。ここまでの定義だけで，辞書型を宣言し，使うことができる。以下に例を示す。

```
Aary dict;
dict["cat"] = "meow";
dict["dog"] = "moo";
dict["cow"] = "woof";
dict["sheep"] = "baa";
dict["dog"] = dict["cow"];
dict["cow"] = "moo";
```

この実装では，インデックス型Iと要素型Eに若干の条件がある。インデックス型には演算子“==”と演算子“=”が定義され，要素型には演算子“=”が定義されている必要がある。さらに，この2つの型は，初期設定子なしで宣言できるか，デフォルトの初期設定子を持っている必要がある。

次に，ここまでの経験を生かし，クラステンプレートを使って総称型を実装することにする。

```
template <class I, class E> Aary;

template <class I, class E>
class Pair {
   I index;
   E el;
   Pair *next;
   friend class Aary<I,E>;
};

template <class I, class E>
class Aary {
  public:
   Aary()
      { elems = 0; }
   ~Aary();
   E &operator []( I );
  private:
   Pair<I,E> *elems;
   Aary( Aary & );
   void operator =( Aary & );
};
```

これで，総称型のインスタンスを生成し，生成した型のオブジェクトを宣言することができる。

以下では，インデックスと要素の型としてStringを使って，辞書型を生成する。

```
typedef Aary<String,String> Dictionary;
Dictionary dict;
```

このほかにも，たとえばコンパイラ記号表

```
Aary<String,NameInfo> symbol_table;
```

あるいは整数のインデックスを持つ非常に大きい疎配列

```
typedef Aary<long,Type> LargeAry;
LargeAry a;
Type x, y, z;
a[0] = x;
a[1000000000] = y;
a[2000000000] = z;
```

を作成することができる。

ただし，連想配列を要素とする連想配列を生成することはできないことに注意されたい。

```
Aary<String,Dictionary> library; // エラー!
```

この理由は何だろう。Aary テンプレートを書き直し，この制限を取り除け。□

演習 4-27 ここでは，直前のリストへのリンクポインタと直後のリストへのリンクポインタをただ 1 つの圧縮ポインタで表現するために，排他的論理和を使った古い秘技を使うことにする。直後の要素のアドレスを保持していれば，圧縮ポインタから直前の要素のアドレスを取り出すことが可能であり，また逆も可能である。

```
encoded = previous ^ next;
next == encoded ^ previous;
previous == encoded ^ next;
```

リストの両方向の走査には，リストの最初と最後の非圧縮アドレスだけが必要である。

```
typedef int Etype;
typedef int Bits;
```

```
class ListEl {
   ListEl *ptr;
   Etype el;
   friend class List;
   friend class Iter;
   friend void paste( ListEl *&, ListEl *&, Etype );
};

class List {
   ListEl *hd, *tl;
  public:
   List() { hd = tl = 0; }
   friend List &operator +( List &, Etype );
   friend List &operator +=( Etype, List & );
   ListEl *head() { return hd; }
   ListEl *tail() { return tl; }
   friend class Iter;
};
```

演算子 “+=” にはリストの最初に要素を追加する操作を，演算子 “+” にはリストの最後に要素を追加する演算を多重定義する。

```
List &
operator +( List &lst, Etype e ) {
   paste( lst.tl, lst.hd, e );
   return lst;
}

List &
operator +=( Etype e, List &lst ) {
   paste( lst.hd, lst.tl, e );
   return lst;
}

void
paste( ListEl *&front, ListEl *&back, Etype e ) {
   ListEl *tmp = new ListEl;
   tmp->el = e;
   if( !front )
      back = front = tmp;
   else if( back == front ) {
      back->ptr = front = tmp;
      tmp->ptr = back;
   }
   else {
      tmp->ptr = front;
      front->ptr =
```

```
        (ListEl *)(Bits(front->ptr) ^ Bits(tmp));
      front = tmp;
    }
}
```

走査の開始時には，リストの最後のアドレスで反復子オブジェクトを初期設定する。

```
class Iter {
   ListEl*prev, *curr;
  public:
   Iter( ListEl *start ) { prev = 0; curr = start; }
   Etype *operator ()();
};

Etype *
Iter::operator ()() {
   if( !curr )
      return 0;
   ListEl *tmp = curr;
   curr = (ListEl *)(Bits(curr->ptr) ^ Bits(prev));
   prev = tmp;
   return &tmp->el;
}
```

これで，リストを作成し両方向に走査するための準備が整った。

```
typedef int Etype;
#include "list.h"

main() {
   int *ip;
   List ints;
   1 += 2 += 3 += ints + 4 + 5 + 6;

   Iter forw = ints.head();
   while( ip = forw() )
      printf( "%d ", *ip );

   Iter backw = ints.tail();
   while( ip = backw() )
      printf( "%d ", *ip );
}
```

この例では，リストに4，5，6，3，2，1の順序で追加したが，リスト中の整数は昇順で並んでいる。この理由は何だろうか。リストの最初に要素を追加するために演算子 “+=” を使ったのはどうしてだろうか。これは優れた考えだろうか。

この解法では，前方向と後方向のリンクの内部表現がリストの利用者に対して隠されていることが重要である。あとで気が変わり，前方向と後方向のリンクを独立したポインタを使うようにしたと変更しても，このリストを利用したコードに影響を与えることはない。□

演習 5-4　最初に，識別子と値を対応付けるために識別子節型とシンボル表を定義する。ここで考える電卓用言語では宣言が不要であるため，識別子が最初に使われると同時に，シンボル表に名前の項目が作成される。

```
class Id : public Node {
  public:
   Id( const char *nm )
      : entry( look( nm ) ) {}
   int set( int i )
      { return entry->value = i; }
   int eval() const
      { return entry->value; }
  private:
   class Nament {
      friend class Id;
      char *name;
      int value;
      Nament *next;
      Nament( const char *nm, Nament *n )
         : name( strcpy( new char[ strlen(nm)+1 ], nm ) ),
           value( 0 ), next( n ) {}
   };
   static Nament *symtab;
   Nament *entry;
   Nament *look( const char * );
};

Id::Nament *
Id::look( const char *nm ) {
   for( Nament *p = symtab; p; p = p->next )
      if( strcmp( p->name, nm ) == 0 )
         return p;
   return symtab = new Nament( nm, symtab );
}
```

次に，代入節型を定義する。

```
class Assign : public Binop {
  public:
   Assign( Id *t, Node *e )
```

```
        : Binop( t, e ) {}
    int eval() const
        { return ((Id *)left)->set( right->eval() ); }
};
```

Assign::evalでは，Id::setを呼び出すためにポインタleftをキャストする必要があることに注意しよう。この理由は何だろうか。Node中のsetを仮想関数として宣言すれば，このキャストを避けることができるが，これはなぜ悪い考えなのだろうか。

通常は，代入演算子を追加する際に，ほかの抽象構文木節型の実装を変更すべきではない。しかし，二項演算子の実装では，評価順序にたいした意味がないという暗黙の仮定を行っているので，Plus::evalの実装を変更する必要がある。

```
int
Plus::eval() const
    { return left->eval() + right->eval(); }
```

C++では，式の評価順序が完全には定義されていないことを思い出していただきたい。したがって，加算の左右の部分木のどちらが最初に評価されるかは実装依存である。評価される式が副作用を持たなければ，先のPlus::evalで通常問題はないが，代入文は副作用を持つので問題が生ずる。“(id = 12) + id”という式で，加算の右の部分木を左の部分木より先に評価したときの結果を考えてみよ。可搬性を高めたPlus::evalの実装を以下に示す。

```
int
Plus::eval() const {
    int l = left->eval();
    return l + right->eval();
}
```

抽象構文木の階層を使った対話的な電卓の実装を以下に示す。

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "Nodes.h"
```

```
class Lex { // 汎用の語彙解析系のインターフェイス
  public:
   Lex( size_t lexeme_size = 81 )
      : str( new char[ lexeme_size ] ) {}
   virtual ~Lex()
      { delete [] str; }
   virtual void scan() = 0;
   int token()
      { return tok; }
   const char *lexeme()
      { return str; }
  protected:
   int tok;
   char *str;
};


class CharLex : public Lex {  // 入力ストリームから文字を読み出す
  protected:
   // (変更されている場合を除いて)標準入力から読み出す
   virtual int nextchar()
      { return getchar(); }
   virtual void unnextchar( char c )
      { ungetc( c, stdin ); }
};

class ExprLex : public CharLex { // 式の語彙解析
  public:
   void scan();
};

enum { ID = 257, INT, EOLN, BAD };

/*
   走査系:
      入力ストリーム中の次のトークンを返す。整数定数と
      識別子に対する属性は次の呼び出し時まで有効である。
*/
void
ExprLex::scan() {
   int c;
   while( 1 )
      switch( c = nextchar() ) {
      case '+': case '-':
      case '*': case '/':
      case '(': case ')':
      case '=':
         tok = c;
         return;
```

```
        case ' ': case '\t':
            continue;
        case '\n':
            tok = EOLN;
            return;
        default:
            if( isdigit( c ) ) {
                char *s = str;
                do
                    *s++ = c;
                while( isdigit( c = nextchar() ) );
                *s = '\0';
                unnextchar( c );
                tok = INT;
                return;
            }
            if( isalpha( c ) ) {
                char *s = str;
                do
                    *s++ = c;
                while( isalnum(c = nextchar()) );
                *s = '\0';
                unnextchar( c );
                tok = ID;
                return;
            }
            tok = BAD;
            return;
        }
}

class Parser { // 汎用の構文解析系のインターフェイス
  public:
   Parser( Lex *lp )
       : lex( lp ) {}
   virtual ~Parser()
       { delete lex; }
   virtual Node *parse() = 0;
  protected:
   void scan()
       { lex->scan(); }
   int token()
       { return lex->token(); }
   const char *lexeme()
       { return lex->lexeme(); }
  private:
   Lex *lex;
};
```

```
/*
   単純数式文法の予測的構文解析系

      E --> T {(+|-)T}
      T --> F {(*|/)F}
      F --> ID | INT | ( E ) | ID = E | -F
*/
class ExprParser : public Parser {
  public :
   ExprParser( Lex *lp )
      : Parser( lp ) {}
   ~ExprParser()
      {}
   Node *parse();
  private:
   Node *e();
   Node *t();
   Node *f();
   void error();
};

Node *
ExprParser::parse() {
   scan();
   Node *ast = e();
   if( token() != EOLN )
      error();
   return ast;
}

void
ExprParser::error() {
   printf( "ERROR!!!\n" );
   exit( 255 );
}

Node *
ExprParser::e() {
   Node *root = t();
   while( 1 )
      switch( token() ) {
      case '+':
         scan();
         root = new Plus( root, t() );
         break;
      case '-':
         scan();
         root = new Minus( root, t() );
         break;
```

```
        default:
            return root;
        }
    }

    Node *
    ExprParser::f() {
        Node *root;
        switch( token() ) {
        case ID:
            root = new Id( lexeme() );
            scan();
            if( token() == '=' ) {
                scan();
                root = new Assign( (Id *)root, e() );
            }
            return root;
        case INT:
            root = new Int( atoi( lexeme() ) );
            scan();
            return root;
        case '(':
            scan();
            root = e();
            if( token() != ')' )
                error();
            scan();
            return root;
        case '-':
            scan();
            return new Uminus( f() );
        default:
            error();
        }
    }

    Node *
    ExprParser::t() {
        Node *root = f();
        while( 1 )
            switch( token() ) {
            case '*':
                scan();
                root = new Times( root, f() );
                break;
            case '/':
                scan();
                root = new Div( root, f() );
                break;
```

```
        default:
            return root;
        }
    }

    /*
        実行部:
            初期設定を行い，非エラーの間は繰り返す
    */
    main() {
        Parser *parser = new ExprParser( new ExprLex );
        while( 1 ) {
            printf( "Enter an expression:  " );
            Node *root = parser->parse();
            if( root ) {
                printf( "Result:  %d\n", root->eval() );
                delete root;
            }
        }
        delete parser;
    }
```

□

演習 5-8 最初に，演習 3-2 の “`void *`” のリストと同様の基底クラスを作成する。

```
class PtrListBase {
  protected:
   PtrListBase();
   ~PtrListBase();
   void prepend( void * );
   void append( void * );
   void *pop_front();
   int empty() const;
  private:
   // ...
};
```

次に，単にポインタの型を検査する目的だけのために，クラステンプレートを作成する。ただし，インライン関数で与えられる型変換は，概してコンパイル時に処理されるので，追加のコードが生成されることは稀である。

```
template <class T>
class PtrList : private PtrListBase {
  public:
   PtrList()
```

```
        {}
    ~PtrList()
        {}
    void prepend( T *t )
        { PtrListBase::prepend( t ); }
    void append( T *t )
        { PtrListBase::append( t ); }
    T *pop_front()
        { return (T *) PtrListBase::pop_front(); }
    int empty() const
        { return PtrListBase::empty(); }
};
```

□

演習 5-10　c）最初に，抽象構文木階層の根のクラスNodeのコンストラクタとデストラクタを修正して，割り当てた節の数を保持するようにする。Nodeから派生したクラスのコンストラクタやデストラクタからこれらは常に呼び出されるため，Nodeのコンストラクタとデストラクタの修正だけで十分である。

```
class Node {
  protected:
   Node()
      { count++; }
  public:
   ~Node()
      { count--; }
   virtual int eval();
   static long num_nodes()
      { return count; }
  private:
   static long count;
};
```

さらに，ほかの場所で静的データメンバcountを定義する。

```
long Node::count = 0;
```

これで，割り当てた節の数を監視する監視型を新たに派生させることができる。

```
class Nalloc : public Monitor {
    long max_nodes;
  public:
   double get_display_value() {
      return Node::num_nodes() > max_nodes
```

```
                  ? max_nodes
                  : Node::num_nodes();
       }
       Nalloc( long max = 1000 ) :
          Monitor( "Nodes", 0, max, 0.5 ),
          max_nodes( max ) {}
};
```

□

演習 5-18　「仮想」コンストラクタの機能を実現する手法の 1 つに，オブジェクトの複製を返すメンバ関数によるものがある。

```
class Thing {
  public:
   virtual Thing *copy() = 0;
   virtual Thing *clone() = 0;
   // ...
};

class OneThing : public Thing {
  public:
   Thing *copy();
   Thing *clone();
   OneThing();
   OneThing( const OneThing & );
   // ...
};

class Another : public Thing {
  public:
   Another( int = 1, int = 2 );
   Thing *copy();
   Thing *clone();
   // ...
  private:
   Another( const Another & );
   // ...
};
```

copy と clone 演算の実装は単純である。

```
Thing *
OneThing::copy()
   { return new OneThing(); }
```

```
Thing *
OneThing::clone()
   { return new OneThing( *this ); }
```

□

演習 5-22 256といった定数の代わりに列挙子TSZが使われた理由は，クラスのメンテナンスを容易にするためである。ハッシュ表のサイズを示すこの定数値は，クラスの実装注の複数の場所に現れるために，これらをすべてを1度に変更できるようにしたほうが便利で安全である。

#defineでは，あるクラススコープの中だけ使うことのできる定数を与えることができない。また，#defineと#undefを駆使してこれを実現することは，先の定数に比べてはるかに難しい。

定数データメンバの値は，コンパイル時にはまだ未定である。また，静的または非静的メンバにかかわらず，データメンバの値は実行時まで不定である。しかし，コンパイラは，クラスのコンパイル時に配列のサイズを知る必要がある。

C++の言語機能のうち，クラススコープで使うことができるコンパイル時定数を与え，さらにメンテナンスが容易であるものは，列挙子だけである。したがって，列挙子が使われるわけである。□

演習 6-4 複数の空港を集中制御するシミュレーション。

```
#include <stdio.h>
#include "task.h"

class Airport;

class Plane : public object {
   static int fltcount;
   long start;
   int fltno;
   Airport *origin;
   Airport *destination;
  public:
   Plane( Airport *o, Airport *d ) {
      fltno = ++fltcount;
      origin = o;
      destination = d;
   }
   long howlong() { return sched::get_clock() - start; }
```

```
      void set() { start = sched::get_clock(); }
      int flt() { return fltno; }
      Airport *to() { return destination; }
      Airport *from() { return origin; }
      void reschedule();
   };

   class PlaneQ {
      qhead *head;
      qtail *tail;
     public:
      PlaneQ() {  head = new qhead( ZMODE, 50 );
                  tail = head->tail();}
      void put( Plane *p ) { p->set();
                  tail->put( (object *)p );}
      Plane *get() { return (Plane *)head->get(); }
      int isroom() { return tail->rdspace(); }
      int notempty() { return head->rdcount(); }
   };

   class AirControl : public task   {
      PlaneQ   *landing, *inair, *incoming, *outgoing;
      PlaneQ   *incircling, *outcircling;
      int ok_take_off;
     public:
      AirControl( PlaneQ *, PlaneQ *, PlaneQ *, PlaneQ * );
      ~AirControl();
      int tok() { return ok_take_off; }
   };

   class GroundControl : public task {
      PlaneQ *takingoff, *onground;
     public:
      GroundControl( PlaneQ*, PlaneQ* );
   };

   class Airport : public task {
      PlaneQ   *takingoff, *landing, *inair,
         *onground, *incoming, *outgoing;
      AirControl *acontrol;
      GroundControl *gcontrol;
      const int index;
      char * const nm;
     public:
      Airport( PlaneQ *, PlaneQ *, int, char * );
      ~Airport();
      char *name() { return nm; }
      friend CentralControl;
   };
```

```
class CentralControl : public task {
   static nairports;
   static Airport **airports;
   static urand *acity;
  public:
   CentralControl();
   ~CentralControl();
   friend Airport *pickacity() {
      return CentralControl::
         airports[CentralControl::acity->draw()];}
};

Airport::Airport( PlaneQ *in, PlaneQ *out,
           int i, char *id ) :
   incoming(in), outgoing(out), index(i), nm(id) {

   takingoff = new PlaneQ;
   inair = new PlaneQ;
   landing = new PlaneQ;
   onground = new PlaneQ;
   acontrol = new AirControl( landing, inair,
            incoming, outgoing );
   gcontrol = new GroundControl( takingoff,onground);
   Plane *tp=0, *lp=0;  // 待機中の飛行機
   int maxwait = 30;
   for(;;){
      delay( 10 );
      if( !lp ) lp = landing->get();
      if( !tp && acontrol->tok() ) tp = takingoff->get();
      if( lp ){
         if( onground->isroom() ) {
            printf("flight %d from %s landing at %s\n",
               lp->flt(), lp->from()->name(), name());
            onground->put( lp );
            lp = 0;
         }
         else if( lp->howlong() > maxwait ) {
            printf("flight %d crash landing at %s!\n",
               lp->flt(), name());
            delete lp;
            lp = 0;
         } else
            printf("flight %d landing at %s delayed\n",
               lp->flt(), name());
      }
      if( tp ) {
         if( inair->isroom() ) {
            printf("flight %d from %s to %s taking off\n",
```

```
                tp->flt(), name(), tp->to()->name());
            inair->put( tp );
            tp = 0;
         } else
            printf("flight %d take off from %s delayed\n",
               lp ->flt(), name() );
      }
   } // for(;;) の終わり
}

AirControl::AirControl( PlaneQ *land, PlaneQ *ina,
            PlaneQ *income, PlaneQ *outgo) :
   landing(land), inair(ina),
            incoming(income), outgoing(outgo) {
   incircling = new PlaneQ;
   outcircling = new PlaneQ;
   Plane *ip = 0, *op = 0;
   ok_take_off = 1;
   for(;;){
      delay( 10 );
      while(inair->notempty() && outcircling->isroom())
         outcircling->put( inair->get() );
      while(incoming->notempty() && incircling->isroom())
         incircling->put( incoming->get() );
      if (!ip ) ip = incircling->get();
      if( !op ) op = outcircling->get();
      if( ip ) {
         if( landing->isroom() ) {
            printf("flight %d arriving at %s\n",
               ip->flt(), ip->to()->name() );
            landing->put( ip );
            ip = 0;
         }
         else {
            printf("flight %d delayed circling %s\n",
               ip->flt(), ip->to()->name() );
         }
      }
      if( op ) {
         if( outgoing->isroom() ) {
            printf("flight %d leaving %s airspace\n",
               op->flt(), op->from()->name() );
            outgoing->put( op );
            ok_take_off = 1;
            op = 0;
         }
         else {
            printf("flight %d delayed circling %s\n",
               op->flt(), op->from()->name() );
```

```
                ok_take_off = 0;
            }
        }
    } // for(;;)の終わり
}

int CentralControl::nairports = 0;
Airport **CentralControl::airports = 0;
urand *CentralControl::acity = 0;

CentralControl::CentralControl() {

    const int n = 10;
    nairports = n;
    acity = new urand ( 0, n-1 );
    static char *nms[ n ] = {
        "New York", "Newark", "Chicago",
        "Denver", "Columbus", "Austin",
        "Dallas", "San Francisco", "Portland",
        "San Jose",
    };
    airports = new Airport*[n];
    for( int i = 0; i < n; i++ ) {
        airports[ i ] =
            new Airport(new PlaneQ,new PlaneQ,i,nms[i]);
    }

    // 各空港に何機かの飛行機を与える
    urand nplanes( 20, 40 );
    for( i = 0; i < n; i++ ) {
        Airport *port = airports[ i ];
        for( int m = nplanes.draw(); m;   m-- )
            port->onground->put( new Plane( 0, port ) );
    }

    // 空港と空港の間の空中の交通を指揮する
    // 空いている空港から新しい飛行機を飛ばす
    for(;;) {
        delay( 10 );
        for( i = 0; i < n; i++ ) {
            Airport *port = airports[ i ];
            Plane *p = port->outgoing->get();
            if( p )
                if( p->to()->incoming->isroom() )
                    p->to()->incoming->put( p );
                else {
                    printf("flight %d crashed",
                        p->flt() );
                    delete p;
```

```
            }
            if( !port->onground->notempty() ) {
               port->onground->put(new Plane(0,port));
            }
         }
   } // for(;;)の終わり
}

AirControl::~AirControl() {
   while( incircling->notempty()
          && incoming->notempty() )
      thistask->delay( 10 );
   Plane *p;
   while( p = outcircling->get() )
      incircling->put( p );
   while( incircling->notempty() )
      thistask->delay( 10 );
   cancel( 0 );
}

GroundControl::GroundControl( PlaneQ *takingoff,
               PlaneQ *onground ) {
   urand n( 10, 30 );
   Plane *p = 0;
   for(;;){
      delay( n.draw() );
      if( !p ) p = onground->get();
      if( p ) {
         if( takingoff->isroom() ) {
            p->reschedule(); // 新しい目的地を設定する
            printf("flight %d leaving %s gate\n",
                  p->flt(), p->from()->name() );
            takingoff->put( p );
            p = 0;
         } else
            printf("flight %d delayed at %s gate\n",
                  p->flt(), p->from()->name() );
      }
   } // for(;;)の終わり
}

int Plane::fltcount = 0;

void
Plane::reschedule() {
   origin = destination;
   destination = pickacity();
   start = sched::get_clock();
}
```

```
Airport::~Airport() {
   gcontrol->cancel(0);
   delete acontrol;
   while( landing->notempty() )
      thistask->delay( 10 );
   printf("%s Airport Closed\n", name() );
   cancel(0);
}

CentralControl::~CentralControl() {
   for( int i = 0; i < nairports ; i++ ) {
      delete airports[ i ];
   }
   printf("CentralControl Closed\n");
   cancel( 0 );
}

main() {
   CentralControl *ccp = new CentralControl;
   thistask->delay( 5000 );
   delete ccp;
   thistask->resultis(0);
}
```

□

演習 7-8 抽象基底クラス`ResourceHandle`は，リスト上の使用中のすべての資源を単に管理し，使用中の各資源に対して仮想デストラクタを実行できるようにする。

```
class ResourceHandle {
  protected:
   ResourceHandle();
  public:
   virtual ~ResourceHandle() = 0;
  protected:
   void *operator new( size_t )
      { return 0; }
   void operator delete( void * )
      {}
  private:
   static ResourceHandle *latest;   // 資源リストの先頭
   ResourceHandle *prev;            // リスト上の１つ前の資源
   static void release();           // すべての資源を解放する
   friend void recover();           // 解放作業が許されるただ１つの関数
};
```

```
ResourceHandle *ResourceHandle::latest = 0;

ResourceHandle::ResourceHandle()
   : prev( latest )
   { latest = this; }

ResourceHandle::~ResourceHandle()
   {}

void
ResourceHandle::release() {
   for( ResourceHandle *p = latest; p; p = p->prev )
      p->~ResourceHandle();
}

void
recover() {
   ResourceHandle::release();
   longjmp( main_loop_env, 1 );
}
```

C++コンパイラで，第 8 章で解説する例外処理機構が利用可能であれば，この厳しいコードよりも，例外処理機構を使うほうがよいだろう。□

演習 7-10 テンプレートを使って総称ポインタ型を作成する。インスタンスの生成にあたっては，ポインタが指すオブジェクトの型を指定する。

```
template <class Rtype>
class Hptr {
   Rtype *p;
  public:
   Hptr( Rtype *v = 0 )
      { p = v; }
   operator Rtype *()
      { return p; }
   Rtype &operator *() {
      check(p);
      return *p;
   }
   Rtype &operator []( long i ) {
      check(p+i);
      return p[i];
   }
   Hptr &operator ++() {
      p++;
      return *this;
   }
```

```
    // etc. ...
};
```

この総称ポインタは，インスタンス生成時に指定された型への通常のポインタとして実装されている。この総称ポインタには，通常のポインタ演算によって，通常のすべてのポインタ算術演算を実装する(ここでは前置演算子“++”の例だけを示した)。演算子“*”と演算子“[]”は，ポインタがフリーストアを指すことを保証するために，間接参照を行う前にポインタアドレスを検査していることに注意してほしい。

この検査関数の実装は，ポインタ型を使う環境に大きく依存する。UNIX 環境では，現在のフリーストアの範囲を調べるためにシステムコールsbrkを使うことができる。

```
static char *ilimit;
class HptrInit {  // 演習 8-16 参照
  public:
   HptrInit() { ilimit = sbrk(0); }
};
static HptrInit init;

void
check( void *p ) {
   if( p < ilimit || p >= sbrk(0) )
      error();
}
```

この代わりに，演算子newとdeleteを再定義すれば，環境独立で，効率のよいものを実現することができる。この手法では，newとdeleteは，フリーストア領域の最初と最後のアドレスのリストを管理し，高機能ポインタが間接参照のたびにcheckを呼び出して，システムコールを発行することを不要にする。

先に定義した総称ポインタ型では，あらゆる型に対するインスタンスを生成することができるので，クラス型だけでなくcharあるいはdoubleといった組み込みの型のへのポインタとしても安全に使うことができる。

```
typedef Hptr<char> CHptr;
typedef Hptr<double> DHptr;

struct Pair {
   int i, j;
};
typedef Hptr<Pair> PHptr;
PHptr pub;
```

演算子“->”は，非クラス型へのポインタには適用できないので，この総称ポインタでは“operator ->”を提供していない。このため，PHptrでPairをアクセスするためには，“PP->i”の代わりに“(*PP).i”という構文を使う必要がある。この総称ポインタ型で演算子“->”を使うことができるように拡張することはできるが，そうすると，新しい総称ポインタ型では非クラス型に対するインスタンスを生成することができなくなってしまう。

この演習の解答を用意する際に，問題をある程度自由に解釈した。フリーストアのメモリだけを指すことを保証する総称ポインタをデザインする予定だったが，フリーストアのメモリだけを**参照する**ポインタとして実装している。このため，ポインタはどんなアドレスでも収納することができ，演算子“*”と演算子“[]”で間接参照されたときに初めてポインタのアドレスが検査される。

今回は，通常のポインタ型への変換のために型変換演算子も用意したので。これが問題とされる可能性がある。

```
char *strcpy( char *, const char * );
CHptr s;
char *t;
strcpy( s, t );
```

型変換演算子を使えば，フリーストアのメモリを指していないアドレスをポインタに収納して使うこともできる。

この問題に対して型変換演算子を取り除くアプローチを取ることもできるが，そうすると今度は，このポインタ型のオブジェクトは，通常のポインタ型で動作するように宣言されている既存の関数や演算子の実引数やオペランドとして使うことができなくなる。これは，この総称ポインタ型の価値を損なうだけでなく，この総称ポインタ型を言語定義の型システムに組み込むことができなくなってしまう。

さらに代案として，間接参照，初期設定，代入，算術演算などの後でポインタの値を検査するようにしても，ポインタが常にフリーストアのメモリを指していることをほぼ保証できる。しかし，データ構造がフリーストアの最後に位置するとき，データ構造の最後を通り越して参照しようとするアルゴリズムに対して，このアプローチはときとして無力である。

```
int
strlen2( CHptr s ) {
   // '\0' までの文字列の長さを返す
   CHptr t = s;
   while( *t++ );
   return t-s;
}
```

ポインタtは，最終的にはsが指す文字列の終わりを通り越したアドレスも格納する。この値は，ときとして，非フリーストアのアドレスであることも想像できるだろう。

これらのアプローチのどちらが正しいか判断できるだろうか。また，これらのアプローチのうち1つを選ぶべき状況があるかどうか判断できるだろうか。□

演習 7-14

```
class Delegand;

class Delegator {
   Delegand *delegand;
  public:
   // クラス Delegator 自体の演算
   void op1();
   // ...
   // ほかのすべての演算はクラス Delegand に委譲される
   Delegand *operator ->()
      { return delegand; }
};

class Delegand {
  public:
   // Delegator と Delegand で共有される演算
   int op1();
   // ...
};
```

この手法では，演算子“.”によってアクセスされるDelegatorのすべての演算の実体は，Delegatorのメンバであり，演算子“->”によってアクセスされるすべての演算の実体は，Delegandのオブジェクトの対応するものに委譲される。

```
extern Delegator d, *dp, &dr;
d.op1();     // Delegator のメンバ
d->op1();    // 委譲される
```

```
dp->op1();  // Delegatorのメンバ
dr->op1();  // 委譲される
```

□

演習 7-15　privateデストラクタを持つオブジェクトを静的オブジェクトあるいは自動オブジェクトとして割り当てることはできない。これは，コンパイラがprivateデストラクタを実行しようとするからである。フリーストアに対しては，オブジェクトを破棄し，デストラクタを実行するための，何らの手段を講じる必要がある。

```
class HeapOnly {
  public:
   HeapOnly();
   void Delete()
      { delete this; }
  private:
   ~HeapOnly();
};
```

このテクニックを使った型は，自動オブジェクトあるいは静的オブジェクトとしてだけ割り当てることができる型と一緒に使われることが多い。たとえば，フリーストアの住民である「実装」オブジェクトが，スタックの住民である「ハンドル」オブジェクトから参照カウンタによって参照されるという使い道が考えられる。

```
class Impl {
   Impl()
      : refs( 1 ) {}
   ~Impl()
      {}
   int refs;
   void func1();
   int func2();
   // ...
   friend class Handle;
};

class Handle {
  public:
   Handle()
      : impl( new Impl ) {}
   Handle( const Handle &h )
      : impl( h.impl )
      { impl->refs++; }
```

```
    ~Handle()
       { if( !--impl->refs ) delete impl; }
    Handle &operator =( const Handle &h ) {
       if( this != &h ) {
          if( !--impl->refs )
             delete impl;
          impl = h.impl;
          impl->refs++;
       }
       return *this;
    }
    Impl *operator ->()
       { return impl; }
   protected:
    Impl *impl;
    void *operator new( size_t )
       { return 0; }
    void operator delete( void * )
       {}
};
```

□

演習 8-1 わかりやすくするために，メッセージバッファ型を作成し，内部の実装バッファへの演算を用意することにする。バッファがあふれたとしてもエラーにせず，デフォルトでは，メッセージが単に切り捨てられる。しかし，メッセージを切り捨てるかどうか，バッファの再割り当てを行い，あふれに対処するかどうかを指定することもできるようにする。

```
class MsgBuf {
  public:
   MsgBuf( size_t length = 81 );
   MsgBuf( const MsgBuf & );
   ~MsgBuf();
   MsgBuf &operator =( const MsgBuf & );  // 深いコピー
   MsgBuf &operator =( const char * );    // 深いコピー
   MsgBuf &operator <<( char );
   MsgBuf &operator <<( int );
   MsgBuf &operator <<( unsigned int );
   MsgBuf &operator <<( long );
   MsgBuf &operator <<( unsigned long );
   MsgBuf &operator <<( float );
   MsgBuf &operator <<( double );
   MsgBuf &operator <<( const char * );
   MsgBuf &operator <<( const MsgBuf & );
```

```
    MsgBuf &operator <<( void (*manip)( MsgBuf & ) );
    operator const char *() const; // 浅いコピー
    int length() const;       // '\0' も数にいれる
    int maxlen() const;       // 現在のバッファサイズ
    void make_expandable();     // あふれ時に再割り当てを行う
    void make_fixed_size();     // 再割り当てを行わない (デフォルト)
    int expanded() const;       // バッファが再割り当てされたかどうか
    int overflowed() const;     // バッファがあふれたかどうか
    friend void clear_buf( MsgBuf & );  // 操作子
  private:
    size_t bmax;    // 現在のバッファ長
    char *b;        // メッセージバッファ
    size_t len;     // 現在の最初の未使用インデックス
    char realloc;   // バッファの拡張許可フラグ
    char overflow;  // バッファあふれフラグ
    int expand();   // バッファの再割り当て
};

inline
MsgBuf::operator const char *() const
    { return b; }

inline int
MsgBuf::length() const
    { return len+1; }

inline int
MsgBuf::maxlen() const
    { return bmax; }

inline void
MsgBuf::make_expandable()
    { realloc = 1; }

inline void
MsgBuf::make_fixed_size()
    { realloc = 0; }

inline int
MsgBuf::expanded() const
    { return realloc ? realloc - 1 : 0; }

inline int
MsgBuf::overflowed() const
    { return overflow; }
```

デフォルトのコンストラクタとデストラクタは，MsgBuf オブジェクトが正しい状態で初期設定され，正しく破壊されることを保証する。

```
MsgBuf::MsgBuf( size_t length )
   : bmax( length < 1 ? 1 : length ), b( new char[bmax] ),
      len( 0 ), realloc( 0 ), overflow( 0 )
   { b[0] = '\0'; }

MsgBuf::~MsgBuf()
   { delete b; }
```

代入演算子は，既存のバッファを破壊し，引数の複製で新しいバッファを初期設定する。効率のために，既存のバッファは，長さが十分であれば，再利用される。

```
MsgBuf &
MsgBuf::operator =( const MsgBuf &m ) {
   return *this = m.b;
}

MsgBuf &
MsgBuf::operator =( const char *s ) {
   if( s != b ) {
      int slen = strlen( s );
      while( bmax < slen )
         if( !expand() ) {
            my_strncpy( b, s, bmax );
            overflow++;
            return *this;
         }
      my_strncpy( b, s, bmax );
      len = slen;
   }
   return *this;
}
```

文字列を挿入する演算子の実装は単純である。

```
MsgBuf &
MsgBuf::operator <<( const char *s ) {
   if( !s )
      return *this;
   int n = strlen( s );
   if( bmax-len > n ) {
stuffit:
      my_strncpy( b+len, s, bmax-len );
      len += n;
   }
   else if( realloc )
      if( expand() )
         goto stuffit;
```

```
        else {
            // できるだけコピーする
stuffsome:
            my_strncpy( b+len, s, bmax-len );
            overflow++;
        }
    else
        goto stuffsome;
    return *this;
}
```

便宜上，初期のうちは，文字列の挿入演算子を使って，ほかの挿入演算子を実装することにする。これらは，あとでもっと効率のよいものに変更する必要があるだろう。

```
MsgBuf &
MsgBuf::operator <<( int i ) {
    char integer[ 12 ];
    sprintf( integer, "%d", i );
    return *this << integer;
}
```

expand 関数は，あふれ時に実装バッファの拡張を処理する。この関数はprivate 関数である。

```
int
MsgBuf::expand() {
    char *newb;
    size_t newmax = (3*bmax)/2;    // 50%増加
    newb = new char[ newmax ];
    my_strncpy( newb, b, newmax );
    b = newb;
    bmax = newmax;
    realloc++;
    delete b;
    return 1;
}
```

ここでは，“operator new” の失敗，スタックあふれ，非同期イベント(たとえば利用者割り込み)といったエラーを捕える試みを行っていない。これらのエラー状態は，システムの正しい機能あるいは通信を経由して処理されると仮定している。たとえば，“operator new” は失敗すると，例外を投げ，アプリケーションはこれを捕える。こういったエラー状態の正しい処理にMsgBuf の関与は要求されない。しかし，MsgBuf も，MsgBuf オブジェクトがスタック上に割り当てられることを要求し，結果として，

例外処理によってデストラクタが実行されたときに，MsgBuf が割り当てたメモリがすべて解放されるようにすることで，回復の手伝いを行うこともある。□

演習 8-2　expand 関数はバッファの再割り当ての役目を持つ。この関数を仮想関数にすることで，クラス派生の際にバッファの再割り当て機能を調整することができる。

```
class MsgBuf {
   // ...
   virtual int expand();
   // ...
};

class PanicOnOvflMsgBuf : public MsgBuf {
   int expand();
};

int
PanicOnOvflMsgBuf::expand() {
   panic();
   return 0;
}
```

□

演習 8-8　複素数の値をまるめる(4 捨 5 入する)関数を利用できると仮定する。

```
complex round( complex );
```

あとは，適用子を適当に提供すればよい。

```
inline complex
operator +( complex val, complex (*manip)(complex) )
   { return manip( val ); }
```

整数の式で同等の機能を提供するには別のアプローチが必要である。利用者定義の演算子は，最低 1 つのクラス型引数を持つ必要がある。int 型はクラス型でないため，“operator +” のもう 1 つの引数にクラス型を持たせる必要がある。以下にその方法を 1 つ紹介しよう。

```
int int_round( int );

class ROUND {};
ROUND iround;
```

```
inline int
operator +( int ival, ROUND /* 飾り */ )
   { return int_round( ival ); }
```

□

演習 8-11　クラスSortedCollectionは正しい順序に並んだ要素のコレクションに関数を適用させる手段を有するため，これを利用して実現しよう。しかし，apply関数には“void (*)(Etype)”型の引数しか渡すことができないので，ostreamをapply関数に渡す手法を考案する必要がある。

```
static ostream *oarg;

void
print( Etype e ) {
   *oarg << e;
   *oarg << " ";
}

ostream &
operator <<( ostream &out, SortedCollection &sc ) {
   out << "{ ";
   oarg = &out;    // つぎあて...
   apply( print );
   out << "}";
   return out;
}
```

この解法に潜在的に存在する問題とは何だろうか。クラスSortedCollectionのインターフェイスを変更できれば，この問題をどのように解決することができるだろうか。制御抽象型は何かの役に立つだろうか。□

演習 8-16　以下に解答の一例を紹介しておく。ライブラリ全体を初期設定するコンストラクタを持つ「初期設定」型を作成し，(ヘッダファイルでなく)ライブラリのソースファイル中で，この型の静的オブジェクトを宣言しておく。

```
// ライブラリのソース中で...
class LibInit {
  public:
   LibInit() {
      // ライブラリの初期設定コード...
   }
```

```
    ~LibInit() {
        // ライブラリの後始末コード...
    }
};
static LibInit init;
```

「初期設定」型の静的オブジェクトを宣言したため，プログラムの実行開始時に，コンストラクタが暗黙のうちに呼び出され，ライブラリが初期設定される。逆に，デストラクタは実行の終了直前に必要とされる後始末を行う。

この代案として，ライブラリのヘッダファイル中で，静的LibInitオブジェクトを定義することもできる。この場合，静的データメンバとして宣言されたカウンタが初期設定と後始末を制御するために使われる。

```
// ライブラリのヘッダ中で...
class LibInit {
  public:
   LibInit() {
      if( !users++ ) {
         // ライブラリの初期設定コード...
      }
   }
   ~LibInit() {
      if( !--users ) {
         // ライブラリの後始末コード...
      }
   }
  private:
   static long users;
};
static LibInit init;

// ライブラリのソース中で...
long LibInit::users = 0;
```

この第2のバージョンは，所定のライブラリが関係する，ほかのライブラリが初期設定されるまで，初期設定を遅らせせることができるという長所を持つ。

しかし，実行時の初期設定の実体は依然として静的オブジェクトであり，複数のファイルでは，静的オブジェクトが初期設定される順番は未定義である。また，コンストラクタを持つクラスの静的オブジェクトも実行時の初期設定の間に初期設定されるのだが，第7章で述べたように，デストラクタの明示的呼び出しとnewの配置構文バージョンを使って，オブジェクトを再び初期設定することも可能である。したがって，こ

れらの静的オブジェクトがLibinit機構によって初期設定された後，デフォルトの実行時の初期設定によって，再び初期設定されることを避けるように，特に注意すべきである。

さらにもう1つのアプローチとして，デフォルトの実行時の初期設定を受け入れ，main関数の実行を開始した後で，独自の初期設定を行うというものが考えられる。

ここでは，ある別のライブラリが先に初期設定されていることを当てにしている，ライブラリ初期設定型の基底クラスとして使うために，Initクラスを定義しよう。Initは，静的初期設定時に，複数の翻訳単位間の依存グラフを構成する。この位相グラフの順番で，各Initオブジェクトの仮想ctorが実行される。また，実行の最後に，各Initオブジェクトの仮想dtorが正しい逆順で実行される。

```
class Init {
  protected:
   Init( Init *a = 0, Init *b = 0,
      Init *c = 0, Init *d = 0 );
   virtual void ctor() = 0;
   virtual void dtor() = 0;
  public:
   virtual ~Init();
   static void globalCtor();
   static void globalDtor();
  private:
   static Init *head;    // Initオブジェクトの大域リスト
   Init *next;           // リスト中の次のInitオブジェクト
   Init *depend[5];      // 依存リスト
   char initted;         // 初期設定が行われたかどうかのフラグ
   char cleanedup;       // 後始末が行われたかどうかのフラグ
   void dfsCtor();
   void dfsDtor();
  protected:
   // フリーストアへの割り当てを禁止する。
   void *operator new( size_t );
   void operator delete( void * );
  private:
   // コピーを禁止する
   void operator =( Init & );
   Init( Init & );
};
```

Initの派生クラスは，一般に，単一の静的オブジェクトを持つ。これらの派生クラス型のオブジェクトに対するコンストラクタとデストラクタは，実行時の静的初期設定時に呼び出され，概して，後で呼び出される初期設定関数のために準備を行う。この

段階では，依存関係にあるほかのライブラリがまだ初期設定されているかどうかが不明であるため，まだ自分のライブラリの初期設定を開始することはできない。

実行時の静的初期設定が完了した後，ライブラリの初期設定を行うために，各Init オブジェクトごとに仮想関数ctorを呼び出すために，静的メンバ関数Init::globalCtorが(概してmain から)呼び出される。同様に，ライブラリの後始末を行うために，各Init オブジェクトごとに仮想関数dtor を呼び出すために，静的メンバ関数Init::globalDtorが呼び出される。

```
Init *Init::head = 0;

Init::Init( Init *a, Init *b, Init *c, Init *d )
   : initted( 0 ), cleanedup( 0 ) {
   depend[0] = a;
   depend[1] = b;
   depend[2] = c;
   depend[3] = d;
   depend[4] = 0;

   // このInitを大域リストに入れる。
   next = head;
   head = this;
}

Init::~Init()
   {}

void
Init::globalCtor() {
   // 依存グラフは接続されている保証はないため，
   // ここでは，すべての節を調べる。
   for( Init *p = head; p; p = p->next )
      p->dfsCtor();
}

void
Init::globalDtor() {
   for( Init *p = head; p; p = p->next )
      p->dfsDtor();
}

void
Init::dfsCtor() {
   if( !initted ) {
      initted = 1;
      for( int i = 0; depend[i]; i++ )
```

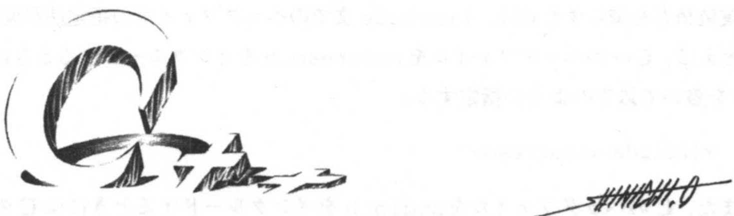
```
            if( !depend[i]->initted )
               depend[i]->dfsCtor();
         ctor();
      }
   }

   void
   Init::dfsDtor() {
      if( !cleanedup ) {
         dtor();
         cleanedup = 1;
         for( int i = 0; depend[i]; i++ )
            if( !depend[i]->cleanedup )
               depend[i]->dfsDtor();
      }
   }
```

ここに示したInit::Initでは，実装を簡単にするために，varargs機能を使う代わりに，依存ファイルの数を4つに制限した。これはかなりの制限であるので，同じ効果を生み出すために，複数のInitオブジェクトを順番に与えるようにすることもできる。□

B

補足



この付録では，C++の標準規格案に追加された新しい機能を紹介する。ただし，現在流通しているC++処理系では，これらの機能はまだ未実装であることが多いことに注意されたい。また，このほかの詳しい仕様に関しては以下の文献を参照されるとよいだろう。

- Margaret A. Ellis and Bjarne Stroustrup,『The Annotated C++ Reference Manual』, Addison-Wesley, 1990
  (邦訳は，トッパンから『注解C++リファレンス・マニュアル』として出版されている)
- Bjarne Stroustrup,『The Design and Evolution of C++』, Addison-Wesley, 1994

この付録は訳者が加筆したものである。

B.1 bool 型

真偽値を表現するためにbool型が追加された。組み込みの定数trueが真値(1)を示し，falseが偽値(0)を示す。これにしたがって，すべての関係演算子(&&，<など)はbool型を返し，制御文(if, for, while, do-while, :?)の条件式はbool値に変換される。

B.2 #include 文

実装依存を減らすために，#include文でのヘッダファイルの指定法が変更された。たとえば，C++のヘッダファイルをiostream.hをインクルードするときには拡張子.hを省いて以下のように指定する。

```
#include <iostream>
```

また，Cのヘッダファイルをstdio.hをインクルードするときにはCのライブラリを示す接頭文字c付けて以下のように指定する。

```
#include <cstdio>
```

B.3 for 文での変数宣言

for 文の 1 番目の節で宣言された変数のスコープは，for 文のブロックの最後までに変更された。例を示す。

```
int i = 0;
for( int i = 0; i < limit ; i++ )  {
   // etc.
}
return i;
```

この場合の i のスコープは for 文の最後で終了するため，return 文での i は 0 である。

B.4 仮想関数の型の制限緩和

派生クラスで，基底クラスの仮想関数をオーバライドする関数の型は，基底クラスの関数のものと厳密に同じであることが要求されていたが，これに関して若干制限が緩和された。例を示す。

ある基底クラス Base と，Base からの public 派生クラス Derived があると仮定すれば，基底クラス B の仮想関数 f の戻り型が “Base *”(あるいは “Base &”)で，派生クラス D の関数の戻り型が “Derived *”(あるいは “Derived &”)であれば，仮想関数のオーバライドが行われる。

```
class B {
   virtual Base f(void);
};

class D {
   Derived f(void);
};
```

したがって，D::f は昔であればエラーであったが，新しい標準規格案では B::f をオーバライドする。

ただし，引数型は厳密に同じであることが依然として要求される。

B.5 配列領域の記憶管理

配列領域の割り当てと解放のために以下の演算子が追加された。

```
void *operator new [](size_t);
void operator delete [](void *);
void operator delete [](void *, size_t);
```

B.6 実行時の型特定機構

C++の仮想関数を使うことによって，複数の派生クラスのオブジェクトを，基底クラスへのポインタでまとめて処理することができる。しかし，基底クラスへのポインタで指し示される先のオブジェクトの型を特定する必要が稀にある。このため，この本当の型を実行時に特定するための機構が追加された。この機構は，**実行時の型情報**(Run-Time Type Information，RTTI)と呼ばれる。

ここでは，§5.4で取りあげたクラスFruitの例を再度使うことにする。基底クラスFruitからの派生クラスとして，Apple，Orangeがあることにする。ただし，これらのクラスは1つ以上の仮想関数を持つこととする。

```
class Fruit { /* ... */ }
class Apple  : public Fruit { /* ... */ }
class Orange : public Fruit { /* ... */ }
```

実行時の型情報はtypeid演算子を使って取得できる。typeid演算子の引数は，ポインタ，リファレンス，オブジェクトあるいは型名であり，実行時の型を表現するtype_info型の大域のconstオブジェクトへのリファレンスを返す。type_info型はヘッダ<typeinfo>で定義される。このtype_infoの戻り値は，演算子==あるいは!=で比較することができる。たとえば，fが“Fruit *”型であるとすると，fが指すオブジェクトの型に応じて，ある処理を指示するためのコードは以下のように表現できる。

```
if (typeid(*f) == typeid(Apple)) {
    /* ... */
}
```

また，type_info型のメンバ関数nameは，自分の型名を返すので，以下のように，fが指すオブジェクトの型を表示することもできる。

```
cout << typeid(*f).name() << endl;
   /* ... */
}
```

この実行時の型情報に関係して，基底クラスへのポインタから派生クラスへのポインタへの型保証キャストを安全に行うためのキャスト演算子dynamic_castも追加された。さらに，Cから継承したキャスト(*type*) *expr*よりも，キャストの意味を明確に指示し，安全にキャストを行うために，C++には新しいキャスト演算子が3つ追加された。以下にこれらの役目をまとめる。

dynamic_cast<*type*>(*expr*)

dynamic_castは，基底クラスへのポインタから派生クラスへのポインタへの型保証キャストを行う。実行時に型の検査が行われ，変換に失敗すれば，0を返す。以下に例を示す。

```
Fruit *pf1 = new Apple;
Fruit *pf2 = new Fruit;

Apple *pa1 = dynamic_cast<Apple *>(pf1);
Apple *pa2 = dynamic_cast<Apple *>(pf2);   /* 失敗 */
```

pa1はAppleのオブジェクトへのポインタであるが，pa2は0である。

const_cast<*type*>(*expr*)

const_castは，constとvolatile修飾子を無効にするだけのキャストを行う。したがってconst_castは，constから非constへ，volatileから非volatileへの変換だけを処理することができる。このほかのときは，コンパイル時にエラーが検出される。

static_cast<*type*>(*expr*)

static_castは，*expr*の型から*type*への暗黙の型変換，あるいは*type*から*expr*の型への暗黙の型変換が存在する場合だけにキャストを行う。このほかのときは，コンパイル時にエラーが検出される。たとえば，intからlongへの変換，派生クラスのポインタから基底クラスのポインタへの変換を処理することができるし，逆にlongからintへの変換，基底クラスのポインタから派生クラスのポインタへの変換，“void *”からの変換を処理することができる。static_castでは，constとvolatile修飾子を無効にする変換はエラーである。

`reinterpret_cast<type>(expr)`

reinterpret_cast は，*type*(*expr*) が許されるのであれば，*expr* を *type* へ単にキャストする。この変換は，C の型変換と同じで，単に型を変更するだけであり，たとえ派生関係があったとしても，ポインタのアドレス自体はキャスト前と同じである。一般に，reinterpret_cast の結果を直接使うことは非常に危険である。static_cast では，const とvolatile 修飾子を無効にする変換はエラーである。

B.7 名前空間

名前空間(namespace)は，大域の関数名，変数名，クラス名の衝突を回避するための機構である。これを使うことで，複数の大域の名前空間を作成し，管理することができる。

新しい名前空間A を作成する例を以下に示す。

```
namespace A {
   int x;
   void f(void);
}
```

同じ名前の名前空間の定義は，同一の名前空間の追加の定義と解釈される。

```
namespace A {   /* ほかの名前空間 A の定義の続き */
   static int y;
   void f(int);
}
```

長い名前空間名を略したいときには，別名を与えることもできる。

```
namespace GraphicLibraryFromHogeComputerService {
   class Point { /* ... */ };
   class Rectangle { /* ... */ };
}
namespace Graphic = GraphicLibraryFromHogeComputerService;
```

名前空間中の名前は，スコープ決定演算子::あるいはusing を使って参照する。以下に，スコープ解決演算子の例を示す。

```
int foo(void) {
   Graphic::Rectangle box;

   A::f(1);
}

int A::y = 100;

A::f(int arg) {
   x = arg;  /* A::x == arg; */
}
```

スコープ決定演算子はすこし冗長である。次に，usingを使って参照することを考える。usingには，using指令とusing宣言の2つがある。先にusing宣言の例を示す。using指令は，特定の名前空間中のすべての名前を現在のスコープで宣言する意味を持つ。

```
int foo(void) {
   using namespace Graphic;
   using namespace A;

   Rectangle box;   /* Graphic::Rectangle box; */
   f(1);            /* A::f(1); */
}
```

using指令は，特定の名前空間中のすべての名前を現在のスコープで宣言する意味を持つので，複数の名前空間をusing指令してしまうと，名前が衝突する可能性が高い。using宣言は，特定の名前空間中の特定の名前を現在のスコープで宣言する。以下に例を示す。

```
int foo(void) {
   using Graphic::Rectangle;
   using A::f;

   Rectangle box;   /* Graphic::Rectangle box; */
   f(1);            /* A::f(1); */
}
```

索引

➡ 参照すべき項目を示す。

［記号索引］

［欧文索引］

［A］

[B]

[C]

[D]

[E]

[F]

[G]

[H]

[I]

[L]

[M]

[N]

[O]

[P]

[R]

[S]

[T]

[U]

[V]

[W]

[X]

[和文索引]

[ア]

[イ]

[エ]

[オ]

[カ]

[ミ]

[メ]

[モ]

[ユ]

[ヨ]

[ラ]

[リ]

著者略歴

Stephen C. Dewhurst

Princeton University にて計算機科学で修士号を取得。AT&T Bell Laboratories で，コンパイラ開発の仕事を中心に働き，Bjarne Stroustrup と知りあう契機であった研究に携わり，C++の仕事を始めた。後に，コンパイラ開発と大学講師といった仕事を経て，現在，ソフトウェアに関する顧問会社である Semantics 社を経営し，妻の Sarah Hewins と 2 歳の息子 David とともに，Cape Cod，Massachusetts 近くの畑の真中で暮らしている。

Kathy T. Stark

State University of New York at Albany にて，計算機科学で修士号を取得。General Electric R&D 社，AT&T Bell Laboratories で働き，現在，Sun Microsystems の SunSoft で働いている。これまでに各種のプログラミングに従事したが，ほとんどはコンパイラとプログラミングツール関係である。現在，同じ Sun の社員である Christopher Aoki とともに，Silicon Valley での California 流の生活を満喫している。

訳者略歴

小山 裕司（こやま ひろし）

東京都立科学技術大学 大学院 工学研究科 修士課程(電子情報系システム工学専攻)修了。現在，東京都立科学技術大学 大学院 工学研究科 博士課程(工学システム専攻 情報制御システム工学)に在籍。著訳書に,「Linux 入門」（アジソン・ウェスレイ・パブリッシャーズ・ジャパン）,「Perl の国にようこそ」(サイエンス社),「注解 C++ リファレンス・マニュアル」(トッパン)などがある。

●本書の内容に関するご質問は、小社スーパーアスキー編集部まで、封書（返信用切手同封のこと）にてお願い致します。
電話によるお問い合わせには、応じられません。
なお、本書の範囲を越える質問に関しては、お答えできない場合もあります。
●落丁・乱丁本は、送料当社負担にてお取り替え致します。
お手数ですが、小社営業部までご返送ください。

改訂新版　C++言語入門

1990年8月21日　初版発行
1995年11月1日　第2版発行

著　者　Stephen C. Dewhurst（ステファン デューハースト）／Kathy T. Stark（キャシー スターク）
翻　訳　小山 裕司（こやま ひろし）
発行人　宮崎 秀規
編集人　土屋 信明
発行所　**株式会社アスキー**
〒151-24　東京都渋谷区代々木4-33-10
振　替　00140-7-161144
大代表　(03)5351-8111
出版営業部 (03)5351-8194（ダイヤルイン）
スーパーアスキー編集部 (03)5351-8125（ダイヤルイン）

印　刷　モリモト印刷株式会社

編　集　金光 雅夫／佐々木 三奈

ISBN4-7561-0320-0　　**Printed in Japan**

●11546／1190231

SOFTWARE SCIENCE

SOFTWARE SCIENCE

ISBN4-7561-0320-0 C3055 P2700E

9784756103208

1913055027009

定価2,700円(本体2,621円)

PROGRAMMING IN C++
SECOND EDITION

郵便はがき

料金受取人払

代々木局承認

1615

差出有効期間
平成9年8月
31日まで
郵便切手は
いりません

151-24

（受取人）

株式会社アスキー
スーパーアスキー編集部 行

フリガナ		年齢	性別
お名前		歳	男・女
ご住所 1. 自宅 2. 勤務先	〒　TEL（　） 都道府県		
勤務先名（学校名）		所属部署名（学科名）	
お勤め先の業種	1. 製造業　2. 金融・保険・不動産業　3. 商社・卸・小売業 4. 建設業　5. 運輸・通信・公共サービス業　6. その他サービス業 7. 政府・公共機関　8. 学校・研究所・病院　9. 農林水産業 10. 情報産業　11. 学生　12. その他		
お勤め先の職種	1. 経営・社業全般　2. 総務・庶務　3. 経理・財務　4. 人事・労務 5. 企画・調査　6. 営業・販売　7. 広報・宣伝　8. 設計・デザイン 9. 資材・購買　10. 生産・工事　11. 研究・開発・技術 12. 情報処理・EDP　13. 教育　14. その他		

改訂新版　C＋＋言語入門

1190231-01

1. 本書は何からお知りになりましたか？

①新聞の広告　②アスキーの雑誌広告　③アスキー以外の雑誌広告
④新聞・雑誌の記事　⑤アスキーの図書目録　⑥知人の紹介　⑦書　店
⑧パソコンショップ　⑨パソコン通信　⑩その他（　　　　）

2. 本書の評価

①内　容　満足　普通　不満　②翻訳の質　満足　普通　不満
③表　紙　満足　普通　不満　④レイアウト　満足　普通　不満
⑤価　格　満足　普通　不満

3. あなたはコンピュータをお使いですか？

①自宅で使っている　→機種名（　　　　）
②会社・学校で使っている　→機種名（　　　　）
③その他で使っている　→機種名（　　　　）
④購入予定がある　→機種名（　　　　）
⑤今後も購入予定はない

4. コンピュータの主な使用目的は何ですか？

①プログラミング　②システム管理　③アプリケーション　④その他

5. ご購読の新聞・雑誌を教えてください。

新　聞（　　　　）
雑　誌（　　　　）

6. ご意見・ご感想がありましたらお聞かせください。

ご協力ありがとうございました。